Lecture Notes in Computer Science 1900

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Arndt Bode Thomas Ludwig
Wolfgang Karl Roland Wismüller (Eds.)

Euro-Par 2000
Parallel Processing

6th International Euro-Par Conference
Munich, Germany, August 29 – September 1, 2000
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Arndt Bode
Thomas Ludwig
Wolfgang Karl
Roland Wismüller
Technische Universität München, Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation, LRR-TUM
80290 München, Deutschland
E-mail: {bode/ludwig/karlw/wismuell}@in.tum.de

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Parallel processing ; proceedings / Euro-Par 2000, 6th International
Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000.
Arndt Bode . . . (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1900)
ISBN 3-540-67956-1

CR Subject Classification (1998): C.1-4, D.1-4, F.1-3, G.1-2, E.1, H.2

ISSN 0302-9743
ISBN 3-540-67956-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH
© Springer-Verlag Berlin Heidelberg 2000
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10722612 06/3142 5 4 3 2 1 0

Preface

Euro-Par – the European Conference on Parallel Computing – is an international conference series dedicated to the promotion and advancement of all aspects of parallel computing. The major themes can be divided into the broad categories of hardware, software, algorithms, and applications for parallel computing. The objective of Euro-Par is to provide a forum within which to promote the development of parallel computing both as an industrial technique and an academic discipline, extending the frontier of both the state of the art and the state of the practice. This is particularly important at a time when parallel computing is undergoing strong and sustained development and experiencing real industrial take up. The main audience for and participants of Euro-Par are seen as researchers in academic departments, government laboratories, and industrial organisations. Euro-Par's objective is to become the primary choice of such professionals for the presentation of new results in their specific areas. Euro-Par is also interested in applications that demonstrate the effectiveness of the main Euro-Par themes.

Euro-Par now has its own Internet domain with a permanent Web site where the history of the conference series is described: `http://www.euro-par.org`. The Euro-Par conference series is sponsored by the Association of Computer Machinery and the International Federation of Information Processing.

Euro-Par 2000

Euro-Par 2000 was organised at the Technische Universität München within walking distance of the newly installed Bavarian Center for Supercomputing HLRB with its Hitachi SR-8000 Teraflop computer at LRZ (Leibniz Rechenzentrum of the Bavarian Academy of Science). The Technische Universität München also hosts the Bavarian Competence Center for High Performance Computing KONWIHR and since 1990 has managed SFB 342 "Tools and Methods for the Use of Parallel Computers" – a large research grant from the German Science Foundation.

The format of Euro-Par 2000 followed that of the preceding five conferences and consists of a number of topics each individually monitored by a committee of four members. There were originally 21 topics for this year's conference. The call for papers attracted 326 submissions of which 167 were accepted. Of the papers accepted, 5 were judged as distinguished, 94 as regular, and 68 as research notes. There were on average 3.8 reviews per paper. Submissions were received from 42 countries, 31 of which were represented at the conference. The principal contributors by country were the United States of America with 29, Germany with 22, U.K. and Spain with 18 papers each, and France with 11 papers. This year's conference, Euro-Par 2000, featured new topics such as Cluster Computing, Metacomputing, Parallel I/O and Storage Technology, and Problem Solving Environments.

Euro-Par 2000 was sponsored by the Deutsche Forschungsgemeinschaft, KONWIHR, the Technische Universität München, ACM, IFIP, the IEEE Task Force on Cluster Computing (TFCC), Force Computers GmbH, Fujitsu-Siemens Computers, Infineon Technologies AG, Dolphin Interconnect Solutions, Hitachi, AEA Technology, the Landeshauptstadt München, Lufthansa, and Deutsche Bahn AG. The conference's Web site is `http://www.in.tum.de/europar2k/`.

Acknowledgments

Organising an international event like Euro-Par 2000 is a difficult task for the conference chairs and the organising committee. Therefore, we are especially grateful to Ron Perrott, Christian Lengauer, Ian Duff, Michel Daydé, and Daniel Ruiz who gave us the benefit of their experience and helped us generously during the 18 months leading up to the conference. The programme committee consisted of nearly 90 members who contributed to the organisation of an excellent academic programme. The programme committee meeting at Munich in April was well attended and, thanks to the sound preparation by everyone and Christian Lengauer's guidance, resulted in a coherent, well-structured conference.

The smooth running and the local organisation of the conference would not have been possible without the help of numerous people. Firstly, we owe special thanks to Elfriede Kelp and Peter Luksch for their excellent work in the organising committee. Secondly, many colleagues were involved in the more technical work. Günther Rackl managed the task of setting up and maintaining our Web site. Georg Acher adapted and improved the software for the submission and refereeing of papers that was inherited from Lyons via Passau, Southampton, and Toulouse. He also spent numerous hours checking and printing the submitted papers. The final papers were handled with the same care by Rainer Buchty. Max Walter prepared the printed programme, Detlef Fliegl helped us in local arrangements. Finally, INTERPLAN Congress, Meeting & Event Management, Munich supported us in the process of registration, hotel reservation, and payment.

June 2000

Arndt Bode
Thomas Ludwig
Wolfgang Karl
Roland Wismüller

Euro-Par Steering Committee

Chair

Ron Perrott	Queen's University Belfast, UK

Vice Chair

Christian Lengauer	University of Passau, Germany

European Representatives

Luc Bougé	ENS Lyon, France
Helmar Burkhart	University of Basel, Switzerland
Péter Kacsuk	MTA SZTAKI Budapest, Hungary
Jeff Reeve	University of Southampton, UK
Henk Sips	Technical University Delft, The Netherlands
Marian Vajtersic	Slovak Academy of Sciences, Slovakia
Mateo Valero	University Polytechnic of Catalonia, Spain
Marco Vanneschi	University of Pisa, Italy
Jens Volkert	University of Linz, Austria
Emilio Zapata	University of Malaga, Spain

Representative of the European Commission

Renato Campo	European Commission, Belgium

Non-european Representatives

Jack Dongarra	University of Tennessee at Knoxville, USA
Shinji Tomita	Kyoto University, Japan

Honorary Member

Karl Dieter Reinartz	University of Erlangen-Nuremberg, Germany

Euro-Par 2000 Local Organisation

Euro-Par 2000 was organised by LRR-TUM (*Lehrstuhl für Rechnertechnik und Rechnerorganisation der Technischen Universität München*), Munich, Germany.

Conference Chairs

Arndt Bode	Thomas Ludwig

Committee

Wolfgang Karl	Elfriede Kelp
Peter Luksch	Roland Wismüller

Technical Committee

Georg Acher	Rainer Buchty
Detlef Fliegl	Günther Rackl
Max Walter	

Euro-Par 2000 Programme Committee

Topic 01: Support Tools and Environments

Global Chair

Barton Miller	University of Wisconsin-Madison, USA

Local Chair

Hans Michael Gerndt	Research Centre Jülich, Germany

Vice Chairs

Helmar Burkhart	University of Basel, Switzerland
Bernard Tourancheau	Laboratoire RESAM, ISTIL, UCB-Lyon, France

Topic 02: Performance Evaluation and Prediction

Global Chair

Thomas Fahringer	University of Vienna, Austria

Local Chair

Wolfgang E. Nagel	Technische Universität Dresden, Germany

Vice Chairs

Arjan J.C. van Gemund	Delft University of Technology, The Netherlands
Allen D. Malony	University of Oregon, USA

Topic 03: Scheduling and Load Balancing

Global Chair

Miron Livny	University of Wisconsin-Madison, USA

Local Chair

Bettina Schnor	University of Potsdam, Germany

Vice Chairs

El-ghazali Talbi	Laboratoire d'Informatique Fondamentale de Lille, France
Denis Trystram	University of Grenoble, France

Topic 04: Compilers for High Performance

Global Chair

Samuel P. Midkiff	IBM, T.J. Watson Research Center, USA

Local Chair

Jens Knoop	Universität Dortmund, Germany

Vice Chairs

Barbara Chapman	University of Houston, USA
Jean-François Collard	Intel Corp., Microcomputer Software Lab.

Topic 05: Parallel and Distributed Databases and Applications

Global Chair

Nelson M. Mattos	IBM, Santa Teresa Laboratory, USA

Local Chair

Bernhard Mitschang	University of Stuttgart, Germany

Vice Chairs

Elisa Bertino	Università di Milano, Italy
Harald Kosch	Universität Klagenfurt, Austria

Topic 06: Complexity Theory and Algorithms

Global Chair

Miroslaw Kutyłowski	University of Wrocław and University of Poznań, Poland

Local Chair

Friedhelm Meyer auf der Heide	University of Paderborn, Germany

Vice Chairs

Gianfranco Bilardi	Dipartimento di Elettronica e Informatic, Padova, Italy
Prabhakar Ragde	University of Waterloo, Canada
Maria José Serna	Universitat Politècnica de Catalunya, Barcelona, Spain

Topic 07: Applications on High-Performance Computers

Global Chair

Jack Dongarra	University of Tennessee, USA

Local Chair

Michael Resch	High Performance Computing Center Stuttgart, Germany

Vice Chairs

Frederic Desprez	INRIA Rhône-Alpes, France
Tony Hey	University of Southampton, UK

Topic 08: Parallel Computer Architecture

Global Chair

Per Stenström	Chalmers University of Technology, Sweden

Local Chair

Silvia Melitta Müller	IBM Deutschland Entwicklung, Labor Böblingen, Germany

Vice Chairs

Mateo Valero	Universidad Politecnica de Catalunya, Barcelona, Spain
Stamatis Vassiliadis	TU Delft, The Netherlands

Topic 09: Distributed Systems and Algorithms

Global Chair

Paul Spirakis	CTI Computer Technology Institute, Patras, Greece

Local Chair

Ernst W. Mayr	TU München, Germany

Vice Chairs

Michel Raynal	IRISA (Université de Rennes and INRIA), France
André Schiper	EPFL, Lausanne, Switzerland
Philippas Tsigas	Chalmers University of Technology, Sweden

Topic 10: Programming Languages, Models, and Methods

Global Chair

Paul H. J. Kelly	Imperial College, London, UK

Local Chair

Sergei Gorlatch	University of Passau, Germany

Vice Chairs

Scott B. Baden	University of California, San Diego, USA
Vladimir Getov	University of Westminster, UK

Topic 11: Numerical Algorithms for Linear and Nonlinear Algebra

Global Chair

Ulrich Rüde	Universität Erlangen-Nürnberg, Germany

Local Chair

Hans-Joachim Bungartz	FORTWIHR, TU München, Germany

Vice Chairs

Marian Vajtersic	Slovak Academy of Sciences, Bratislava, Slovakia
Stefan Vandewalle	Katholieke Universiteit Leuven, Belgium

Topic 12: European Projects

Global Chair

Roland Wismüller	TU München, Germany

Vice Chair

Renato Campo	European Commission, Bruxelles, Belgium

Topic 13: Routing and Communication in Interconnection Networks

Global Chair

Manolis G. H. Katevenis	University of Crete, Greece

Local Chair

Michael Kaufmann	Universität Tübingen, Germany

Vice Chairs

Jose Duato	Universidad Politecnica de Valencia, Spain
Danny Krizanc	Wesleyan University, USA

Topic 14: Instruction-Level Parallelism and Processor Architecture

Global Chair

Kemal Ebcioğlu	IBM, T.J. Watson Research Center, USA

Local Chair

Theo Ungerer	University of Karlsruhe, Germany

Vice Chairs

Nader Bagherzadeh	University of California, Irvine, USA
Mariagiovanna Sami	Politecnico di Milano, Italy

Topic 15: Object Oriented Architectures, Tools, and Applications

Global Chair

Gul A. Agha	University of Illinois at Urbana-Champaign, USA

Local Chair

Michael Philippsen	Universität Karlsruhe, Germany

Vice Chairs

Françoise Baude	I3S/INRIA, Sophia Antipolis, France
Uwe Kastens	Universität-GH Paderborn, Germany

Topic 16: High Performance Data Mining and Knowledge Discovery

Global Chair

David B. Skillicorn	Queen's University, Kingston, Canada

Local Chair

Kilian Stoffel	Institut Interfacultaire d'Informatique, Neuchâtel, Switzerland

Vice Chairs

Arno Siebes	Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
Domenico Talia	ISI-CNR, Rende, Italy

Topic 17: Architectures and Algorithms for Multimedia Applications

Global Chair

Andreas Uhl	University of Salzburg, Austria

Local Chair

Manfred Schimmler	TU Braunschweig, Germany

Vice Chairs

Pieter P. Jonker	Delft University of Technology, The Netherlands
Heiko Schröder	School of Applied Science, Singapore

Topic 18: Cluster Computing

Global Chair

Rajkumar Buyya	Monash University, Melbourne, Australia

Local Chair

Djamshid Tavangarian	University of Rostock, Germany

Vice Chairs

Mark Baker	University of Portsmouth, UK
Daniel C. Hyde	Bucknell University, USA

Topic 19: Metacomputing

Global Chair

Geoffrey Fox	Syracuse University, USA

Local Chair

Alexander Reinefeld	Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany

Vice Chairs

Domenico Laforenza	Institute of the Italian National Research Council, Pisa, Italy
Edward Seidel	Albert-Einstein-Institut, Golm, Germany

Topic 20: Parallel I/O and Storage Technology

Global Chair

Rajeev Thakur	Argonne National Laboratory, USA

Local Chair

Peter Brezany	University of Vienna, Austria

Vice Chairs

Rolf Hempel	NEC Europe, Germany
Elizabeth Shriver	Bell Laboratories, USA

Topic 21: Problem Solving Environments

Global Chair

José C. Cunha	Universidade Nova de Lisboa, Portugal

Local Chair

Wolfgang Gentzsch	GRIDWARE GmbH & Inc., Germany

Vice Chairs

Thierry Priol	IRISA/INRIA, Paris, France
David Walker	Oak Ridge National Laboratory, USA

Euro-Par 2000 Referees

(not including members of the programme and organisation committees)

Aburdene, Maurice
Adamidis, Panagiotis
Alcover, Rosa
Almasi, George
Alpern, Bowen
Alpert, Richard
Altman, Erik
Ancourt, Corinne
Aoun, Mondher Ben
Arpaci-Dusseau, A.
Avermiddig, Alfons
Ayguade, Eduard
Azevedo, Ana
Bailly, Arnaud
Bampis, E.
Barrado, Cristina
Barthou, Denis
Basu, Sujoy
Beauquier, Joffroy
Becker, Juergen
Becker, Wolfgang
Beckmann, Olav
Beivide, Ramón
Bellotti, Francesco
Benger, Werner
Benkner, Siegfried
Benner, Peter
Bernard, Toursel
Berrendorf, Rudolf
Berthou, Jean-Yves
Bettati, Riccardo
Bhandarkar, Suchendra
Bianchini, Ricardo
Bischof, Holger
Blackford, Susan
Bodin, Francois
Bordawekar, Rajesh
Bose, Pradip
Boulet, Pierre
Brandes, Thomas
Bregier, Frederic
Breshears, Clay
Breveglieri, Luca
Brieger, Leesa
Brinkschulte, Uwe
Brorsson, Mats
Brunst, Holger
Brzezinski, Jerzy
Burton, Ariel
Cachera, David
Carpenter, Bryan
Casanova, Henri
Chakravarty, Manuel M. T.
Charles, Henri-Pierre
Charney, Mark
Chassin de Kergommeaux, Jacques
Chatterjee, Siddhartha
Cheng, Benny
Chlebus, Bogdan
Chou, Pai
Christodorescu, Mihai
Christoph, Alexander
Chung, Yongwha
Chung, Yoo C.
Cilio, Andrea
Citron, Daniel
Clauss, Philippe
Clement, Mark
Coelho, Fabien
Cohen, Albert
Cole, Murray
Contassot-Vivier, Sylvain
Cortes, Toni
Cosnard, Michel
Costen, Fumie
Cox, Simon
Cozette, Olivier
Cremonesi, Paolo
Czerwinski, Przemyslaw
Damper, Robert
Dang, Minh
Dekeyser, Jean-Luc

Dhaenens, Clarisse
Di Martino, Beniamino
Diks, Krzysztof
Dimakopoulos, Vassilios V.
Dini, Gianluca
Djelic, Ivan
Djemai, Kebbal
Domas, Stephane
Downey, Allen
Dubois, Michel
Duchien, Laurence
Dumitrescu, Bogdan
Egan, Colin
Eisenbeis, Christine
Ekanadham, Kattamuri
Ellmenreich, Nils
Elmroth, Erik
Emer, Joel
Espinosa, Antonio
Etiemble, Daniel
Faber, Peter
Feitelson, Dror
Fenwick, James
Feo, John
Fernau, Henning
Ferstl, Fritz
Field, Tony
Filho, Eliseu M. Chaves
Fink, Stephen
Fischer, Bernd
Fischer, Markus
Fonlupt, Cyril
Formella, Arno
Fornaciari, William
Foster, Ian
Fragopoulou, Paraskevi
Friedman, Roy
Fritts, Jason
Frumkin, Michael
Gabber, Eran
Gaber, Jaafar
Garatti, Marco
Gatlin, Kang Su
Gaudiot, Jean-Luc
Gautama, Hasyim
Gautier, Thierry
Geib, Jean-Marc
Genius, Daniela
Geoffray, Patrick
Gerbessiotis, Alexandros
Germain-Renaud, Cecile
Giavitto, Jean-Louis
Glamm, Bob
Glaser, Hugh
Glendinning, Ian
Gniady, Chris
Goldman, Alfredo
Golebiewski, Maciej
Gomes, Cecilia
Gonzalez, Antonio
Gottschling, Peter
Graevinghoff, Andreas
Gray, Paul
Grewe, Claus
Griebl, Martin
Grigori, Laura
Gschwind, Michael
Guattery, Steve
Gubitoso, Marco
Guerich, Wolfgang
Guinand, Frederic
v. Gudenberg, Juergen Wolff
Gupta, Manish
Gupta, Rajiv
Gustavson, Fred
Gutheil, Inge
Haase, Gundolf
Hall, Alexander
Hallingström, Georg
Hammami, Omar
Hańćkowiak, Michał
Hank, Richard
Hartel, Pieter
Hartenstein, Reiner
Hatcher, Phil
Haumacher, Bernhard
Hawick, Ken
Heath, James
Heiss, Hans-Ulrich
Hélary, Jean-Michel

Helf, Clemens
Henzinger, Tom
Herrmann, Christoph
Heun, V.
Hind, Michael
Hochberger, Christian
Hoeflinger, Jay
Holland, Mark
Hollingsworth, Jeff
Holmes, Neville
Holzapfel, Klaus
Hu, Zhenjiang
Huckle, Thomas
Huet, Fabrice
d'Inverno, Mark
Irigoin, Francois
Jacquet, Jean-Marie
Jadav, Divyesh
Jamali, Nadeem
Janzen, Jens
Jay, Barry
Jeannot, Emmanuel
Ji, Minwen
Jiang, Jianmin
Jin, Hai
Johansson, Bengt
Jonkers, Henk
José Serna, Maria
Jung, Matthias T.
Jung, Michael
Jurdziński, Tomasz
Juurlink, Ben
Kaiser, Timothy
Kaklamanis, Christos
Kallahalla, Mahesh
Kanarek, Przemysława
Karavanic, Karen L.
Karlsson, Magnus
Karner, Herbert
Kavi, Krishna
Kaxiras, Stefanos
Keller, Gabriele
Keller, Joerg
Kessler, Christoph
Khoi, Le Dinh
Kielmann, Thilo
Klauer, Bernd
Klauser, Artur
Klawonn, Axel
Klein, Johannes
Knottenbelt, William
Kohl, James
Kolla, Reiner
Konas, Pavlos
Kosch, Harald
Kowaluk, Mirosław
Kowarschik, Markus
Krakowski, Christian
Kralovic, Rastislav
Kranzlmüller, Dieter
Kremer, Ulrich
Kreuzinger, Jochen
Kriaa, Faisal
Krishnaiyer, Rakesh
Krishnamurthy, Arvind
Krishnan, Venkata
Krishnaswamy, Vijaykumar
Kshemkalyani, Ajay
Kuester, Uwe
Kugelmann, Bernd
Kunde, Manfred
Kunz, Thomas
Kurc, Tahsin
Kutrib, Martin
Lageweg, Casper
Lancaster, David
von Laszewski, Gregor
Laure, Erwin
Leary, Stephen
Lechner, Ulrike
Lee, Jaejin
Lee, Wen-Yen
Lefevre, Laurent
Lepere, Renaud
Leupers, Rainer
Leuschel, Michael
Liedtke, Jochen
van Liere, Robert
Lim, Amy
Lin, Fang-Pang

Lin, Wenyen
Lindenmaier, Goetz
Lipasti, Mikko
Liskiewicz, Maciej
Litaize, Daniel
Liu, Zhenying
Lonsdale, Guy
Loogen, Rita
Lopes, Cristina
Lopez, Pedro
Löwe, Welf
Lu, Paul
Lueling, Reinhard
Luick, Dave
Luick, David
Lupu, Emil
Lusk, Ewing
Macêdo, Raimundo
Madhyastha, Tara
Magee, Jeff
Manderson, Robert
Margalef, Tomas
Markatos, Evangelos
Massari, Luisa
Matsuoka, Satoshi
May, John
McKee, Sally
McLarty, Tyce
Mehaut, Jean-Francois
Mehofer, Eduard
Mehrotra, Piyush
Merlin, John
Merz, Stephan
Merzky, Andre
Meunier, Francois
Meyer, Ulrich
Mohr, Bernd
Mohr, Marcus
Montseny, Eduard
Moore, Ronald
More, Sachin
Moreira, Jose
Moreno, Jaime
Mounie, Gregory
Muller, Gilles
Müller-Olm, Markus
Mussi, Philippe
Nair, Ravi
Naroska, Edwin
Navarro, Carlos
Newhouse, Steven
Nicole, Denis A.
Niedermeier, Rolf
Nilsson, Henrik
Nilsson, Jim
Nordine, Melab
O'Boyle, Michael
Ogston, Elizabeth
Ohmacht, Martin
Oldfield, Ron
Oliker, Leonid
Oliveira, Rui
Olk, Eddy
Omondi, Amos
Oosterlee, Cornelis
Osterloh, Andre
Otoo, Ekow
Pan, Yi
Papatriantafilou, Marina
Papay, Juri
Parigot, Didier
Parmentier, Gilles
Passos, Nelson L.
Patras, Ioannis
Pau, Danilo Pietro
Pedone, Fernando
Pelagatti, Susanna
Penz, Bernard
Perez, Christian
Petersen, Paul
Petiton, Serge
Petrini, Fabrizio
Pfahler, Peter
Pfeffer, Matthias
Pham, CongDuc
Philippas, Tsigas
Pimentel, Andy D.
Pingali, Keshav
Pinkston, Timothy
Piotrów, Marek

Pizzuti, Clara
Plata, Oscar
Pleisch, Stefan
Pollock, Lori
Poloni, Carlo
Pontelli, Enrico
Pouwelse, Johan
Požgaj, Aleksandar
Prakash, Ravi
Primet, Pascale
Prvulovic, Milos
Pugh, William
Quaglia, Francesco
Quiles, Francisco
Raab, Martin
Rabenseifner, Rolf
Rackl, Günther
Radulescu, Andrei
Raje, Rajeev
Rajopadhye, Sanjay
Ramet, Pierre
Ramirez, Alex
Rana, Omer
Randriamaro, Cyrille
Rastello, Fabrice
Rau, B. Ramakrishna
Rau, Bob
Rauber, Thomas
Rauchwerger, Lawrence
Reeve, Jeff
Reischuk, Rüdiger
Rieping, Ingo
Riley, Graham
Rinard, Martin
Ripeanu, Matei
Rips, Sabina
Ritter, Norbert
Robles, Antonio
Rodrigues, Luis
Rodriguez, Casiano
Roe, Paul
Roman, Jean
Roos, Steven
Ross, Robert
Roth, Philip
Rover, Diane
Ruenger, Gudula
Rundberg, Peter
Rüthing, Oliver
Sathaye, Sumedh
Sayrs, Brian
van der Schaaf, Arjen
Schaeffer, Jonathan
Schickinger, Thomas
Schikuta, Erich
Schmeck, Hartmut
Schmidt, Bertil
Schöning, Harald
Schreiber, Rob
Schreiner, Wolfgang
Schuele, Josef
Schulz, Joerg
Schulz, Martin
Schwiegelshohn, Uwe
Scott, Stephen L.
Scurr, Tony
Seidl, Stephan
Serazzi, Giuseppe
Serpanos, Dimitrios N.
Sethu, Harish
Setz, T.
Sevcik, Kenneth
Shen, Hong
Shen, John
Shende, Sameer
Sibeyn, Jop F.
Sigmund, Ulrich
Silva, Joao Gabriel
Silvano, Cristina
Sim, Leo Chin
Simon, Beth
Singh, Hartej
Skillicorn, David
Skipper, Mark
Smirni, Evgenia
Soffa, Mary-Lou
Sohler, Christian
Speirs, Neil
Stachowiak, Grzegorz
Stals, Linda

Stamoulis, George
Stathis, Pyrrhos
Staudacher, Jochen
Stefan, Petri
Stefanovic, Darko
Steffen, Bernhard
Steinmacher-Burow, Burkhard D.
Stéphane, Ubeda
Stoodley, Mark
van Straalen, Brian
Striegnitz, Joerg
Strout, Michelle
Struhar, Milan
von Stryk, Oskar
Su, Alan
Sun, Xian-He
de Supinski, Bronis R.
Suri, Neeraj
Suter, Frederic
Sykora, Ondrej
Tan, Jeff
Tanaka, Yoshio
Tchernykh, Andrei
Teck Ng, Wee
Teich, Jürgen
Tessera, Daniele
Thati, Prasannaa
Thies, Michael
Thiruvathukal, George K.
Thomas, Philippe
Thomasset, François
Tixeuil, Sebastien
Tomsich, Philipp
Topham, Nigel
Traff, Jesper Larsson
Trefethen, Anne
Trinitis, Jörg
Tseng, Chau-Wen
Tullsen, Dean
Turek, Stefan
Unger, Andreas
Unger, Herwig
Utard, Gil
Valero-Garcia, Miguel
Varela, Carlos
Varvarigos, Emmanouel
Varvarigou, Theodora
Vayssiere, Julien
Vazhkudai, Sudharshan
Villacis, Juan
Vitter, Jeff
Vrto, Imrich
Waldschmidt, Klaus
Wang, Cho-Li
Wang, Ping
Wanka, Rolf
Watson, Ian
Weimar, Joerg R.
Weiss, Christian
Weisz, Willy
Welch, Peter
Welsh, Matt
Werner, Andreas
Westrelin, Roland
Wilhelm, Uwe
Wirtz, Guido
Wisniewski, Len
Wolski, Rich
Wong, Stephan
Wonnacott, David
Wylie, Brian J. N.
Xhafa, Fatos
Xue, Jingling
Yan, Jerry
Yeh, Chihsiang
Yeh, Tse-Yu
Yelick, Katherine
Yew, Pen-Chung
Zaslavsky, Arkady
Zehendner, Eberhard
Zhang, Yi
Zhang, Yong
Ziegler, Martin
Zilken, Herwig
Zimmer, Stefan
Zimmermann, Falk
Zimmermann, Wolf
Zosel, Mary
Zumbusch, Gerhard

Table of Contents

Invited Talks

Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations 1
David E. Keyes

E2K Technology and Implementation 18
Boris Babayan

Grid-Based Asynchronous Migration of Execution Context in Java Virtual Machines 22
Gregor von Laszewski, Kazuyuki Shudo, Yoichi Muraoka

Logical Instantaneity and Causal Order: Two "First Class" Communication Modes for Parallel Computing 35
Michel Raynal

The TOP500 Project of the Universities Mannheim and Tennessee 43
Hans Werner Meuer

Topic 01

Support Tools and Environments 45
Barton P. Miller, Michael Gerndt

Visualization and Computational Steering in Heterogeneous Computing Environments 47
Sabine Rathmayer

A Web-Based Finite Element Meshes Partitioner and Load Balancer . . . 57
Ching-Jung Liao

A Framework for an Interoperable Tool Environment *(Research Note)* . . 65
Radu Prodan, John M. Kewley

ToolBlocks: An Infrastructure for the Construction of Memory Hierarchy Analysis Tools *(Research Note)* 70
Timothy Sherwood, Brad Calder

A Preliminary Evaluation of FINESSE, a Feedback-Guided Performance Enhancement System 75
Nandini Mukherjee, Graham D. Riley, John R. Gurd

On Combining Computational Differentiation and Toolkits for Parallel Scientific Computing 86
Christian H. Bischof, H. Martin Bücker, Paul D. Hovland

Generating Parallel Program Frameworks from Parallel Design Patterns 95
Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling

Topic 02
Performance Evaluation and Prediction 105
Thomas Fahringer, Wolfgang E. Nagel

A Callgraph-Based Search Strategy for Automated Performance Diagnosis *(Distinguished Paper)* 108
Harold W. Cain, Barton P. Miller, Brian J.N. Wylie

Automatic Performance Analysis of MPI Applications Based on Event Traces 123
Felix Wolf, Bernd Mohr

Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions 133
Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein

A Statistical-Empirical Hybrid Approach to Hierarchical Memory Analysis 141
Xian-He Sun, Kirk W. Cameron

Use of Performance Technology for the Management of Distributed Systems 149
Darren J. Kerbyson, John S. Harper, Efstathios Papaefstathiou, Daniel V. Wilcox, Graham R. Nudd

Delay Behavior in Domain Decomposition Applications 160
Marco Dimas Gubitoso, Carlos Humes Jr.

Automating Performance Analysis from UML Design Patterns *(Research Note)* 168
Omer F. Rana, Dave Jennings

Integrating Automatic Techniques in a Performance Analysis Session *(Research Note)* 173
Antonio Espinosa, Tomas Margalef, Emilio Luque

Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Efficient On-the-Fly Simulation *(Research Note)* 178
Thierry Lafage, André Seznec

SCOPE - The Specific Cluster Operation and Performance Evaluation Benchmark Suite *(Research Note)* 183
Panagiotis Melas, Ed J. Zaluska

Implementation Lessons of Performance Prediction Tool for Parallel Conservative Simulation *(Research Note)* 189
Chu-Cheow Lim, Yoke-Hean Low, Boon-Ping Gan, Wentong Cai

A Fast and Accurate Approach to Analyze Cache Memory Behavior *(Research Note)* 194
Xavier Vera, Josep Llosa, Antonio González, Nerina Bermudo

Impact of PE Mapping on Cray T3E Message-Passing Performance 199
Eduardo Huedo, Manuel Prieto, Ignacio M. Llorente, Francisco Tirado

Performance Prediction of a NAS Benchmark Program with ChronosMix Environment 208
Julien Bourgeois, François Spies

Topic 03
Scheduling and Load Balancing 217
Bettina Schnor

A Hierarchical Approach to Irregular Problems *(Research Note)* 218
Fabrizio Baiardi, Primo Becuzzi, Sarah Chiti, Paolo Mori, Laura Ricci

Load Scheduling with Profile Information 223
Götz Lindenmaier, Kathryn S. McKinley, Olivier Temam

Neighbourhood Preserving Load Balancing: A Self-Organizing Approach 234
Attila Gürsoy, Murat Atun

The Impact of Migration on Parallel Job Scheduling for Distributed Systems 242
Yanyong Zhang, Hubertus Franke, Jose E. Moreira, Anand Sivasubramaniam

Memory Management Techniques for Gang Scheduling 252
William Leinberger, George Karypis, Vipin Kumar

Exploiting Knowledge of Temporal Behaviour in Parallel Programs for Improving Distributed Mapping 262
Concepció Roig, Ana Ripoll, Miquel A. Senar, Fernando Guirado, Emilio Luque

Preemptive Task Scheduling for Distributed Systems *(Research Note)* . . 272
Andrei Rădulescu, Arjan J.C. van Gemund

Towards Optimal Load Balancing Topologies 277
Thomas Decker, Burkhard Monien, Robert Preis

Scheduling Trees with Large Communication Delays on Two Identical Processors 288
Foto Afrati, Evripidis Bampis, Lucian Finta, Ioannis Milis

Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning *(Distinguished Paper)* 296
Kirk Schloegel, George Karypis, Vipin Kumar

Experiments with Scheduling Divisible Tasks in Clusters of Workstations 311
Maciej Drozdowski, Paweł Wolniewicz

Optimal Mapping of Pipeline Algorithms *(Research Note)* 320
Daniel González, Francisco Almeida, Luz Marina Moreno, Casiano Rodríguez

Dynamic Load Balancing for Parallel Adaptive Multigrid Solvers with Algorithmic Skeletons *(Research Note)* 325
Thomas Richert

Topic 04
Compilers for High Performance 329
Samuel P. Midkiff, Barbara Chapman, Jean-François Collard, Jens Knoop

Improving the Sparse Parallelization Using Semantical Information at Compile-Time 331
Gerardo Bandera, Emilio L. Zapata

Automatic Parallelization of Sparse Matrix Computations : A Static Analysis 340
Roxane Adle, Marc Aiguier, Franck Delaplace

Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition 349
Rashindra Manniesing, Ireneusz Karkowski, Henk Corporaal

Temporary Arrays for Distribution of Loops with Control Dependences 357
Alain Darte, Georges-André Silber

Automatic Generation of Block-Recursive Codes 368
Nawaaz Ahmed, Keshav Pingali

Left-Looking to Right-Looking and Vice Versa: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring 379
Nikolay Mateev, Vijay Menon, Keshav Pingali

Identifying and Validating Irregular Mutual Exclusion Synchronization in Explicitly Parallel Programs *(Research Note)* 389
Diego Novillo, Ronald C. Unrau, Jonathan Schaeffer

Exact Distributed Invalidation 395
Rupert W. Ford, Michael F.P. O'Boyle, Elena A. Stöhr

Scheduling the Computations of a Loop Nest with Respect to a Given Mapping 405
Alain Darte, Claude Diderich, Marc Gengler, Frédéric Vivien

Volume Driven Data Distribution for NUMA-Machines 415
Felix Heine, Adrian Slowik

Topic 05
Parallel and Distributed Databases and Applications 425
Bernhard Mitschang

Database Replication Using Epidemic Communication 427
JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi

Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases 435
Klemens Böhm, Torsten Grabs, Uwe Röhm, Hans-Jörg Schek

A Communication Infrastructure for a Distributed RDBMS *(Research Note)* 445
Michael Stillger, Dieter Scheffner, Johann-Christoph Freytag

Distribution, Replication, Parallelism, and Efficiency Issues in a Large-Scale Online/Real-Time Information System for Foreign Exchange Trading *(Research Note)* 451
Peter Peinl

Topic 06
Complexity Theory and Algorithms 455
Friedhelm Mayer auf der Heide, Mirosław Kutyłowski, Prabhakar Ragde

Positive Linear Programming Extensions: Parallel Complexity and Applications *(Research Note)* 456
Pavlos S. Efraimidis, Paul G. Spirakis

Parallel Shortest Path for Arbitrary Graphs 461
Ulrich Meyer, Peter Sanders

Periodic Correction Networks 471
Marcin Kik

Topic 07

Applications on High-Performance Computers **479**
Michael Resch

An Efficient Algorithm for Parallel 3D Reconstruction of Asymmetric Objects from Electron Micrographs 481
Robert E. Lynch, Hong Lin, Dan C. Marinescu

Fast Cloth Simulation with Parallel Computers 491
Sergio Romero, Luis F. Romero, Emilio L. Zapata

The Input, Preparation, and Distribution of Data for Parallel GIS Operations *(Research Note)* 500
Gordon J. Darling, Terence M. Sloan, Connor Mulholland

Study of the Load Balancing in the Parallel Training for Automatic Speech Recognition *(Research Note)* 506
El Mostafa Daoudi, Pierre Manneback, Abdelouafi Meziane, Yahya Ould Mohamed El Hadj

Pfortran and Co-Array Fortran as Tools for Parallelization of a Large-Scale Scientific Application 511
Piotr Bała, Terry W. Clark

Sparse Matrix Structure for Dynamic Parallelisation Efficiency 519
Markus Ast, Cristina Barrado, José Cela, Rolf Fischer, Jesús Labarta, Óscar Laborda, Hartmut Manz, Uwe Schulz

A Multi-color Inverse Iteration for a High Performance Real Symmetric Eigensolver *(Research Note)* 527
Ken Naono, Yusaku Yamamoto, Mitsuyoshi Igai, Hiroyuki Hirayama, Nobuhiro Ioki

Parallel Implementation of Fast Hartley Transform (FHT) in Multiprocessor Systems *(Research Note)* 532
Felicia Ionescu, Andrei Jalba, Mihail Ionescu

Topic 08

Parallel Computer Architecture **537**
Silvia Müller, Per Stenström, Mateo Valero, Stamatis Vassiliadis

Coherency Behavior on DSM: A Case Study *(Research Note)* 539
Jean-Thomas Acquaviva, William Jalby

Hardware Migratable Channels *(Research Note)* 545
David May, Henk Muller, Shondip Sen

Reducing the Replacement Overhead on COMA Protocols for Workstation-Based Architectures 550
Diego R. Llanos Ferraris, Benjamín Sahelices Fernández, Agustín De Dios Hernández

Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs 558
Aleksandar Milenkovic, Veljko Milutinovic

Adaptive Proxies: Handling Widely-Shared Data in Shared-Memory Multiprocessors *(Research Note)* 567
Sarah A.M. Talbot, Paul H.J. Kelly

Topic 09
Distributed Systems and Algorithms **573**
Ernst W. Mayr

A Combinatorial Characterization of Properties Preserved by Antitokens 575
Costas Busch, Neophytos Demetriou, Maurice Herlihy, Marios Mavronicolas

Searching with Mobile Agents in Networks with Liars 583
Nicolas Hanusse, Evangelos Kranakis, Danny Krizanc

Complete Exchange Algorithms for Meshes and Tori Using a Systematic Approach *(Research Note)* 591
Luis Díaz de Cerio, Miguel Valero-García, Antonio González

Algorithms for Routing AGVs on a Mesh Topology *(Research Note)* 595
Ling Qiu, Wen-Jing Hsu

Self-Stabilizing Protocol for Shortest Path Tree for Multi-cast Routing in Mobile Networks *(Research Note)* 600
Sandeep K.S. Gupta, Abdelmadjid Bouabdallah, Pradip K. Srimani

Quorum-Based Replication in Asynchronous Crash-Recovery Distributed Systems *(Research Note)* 605
Luís Rodrigues, Michel Raynal

Timestamping Algorithms: A Characterization and a Few Properties . . . 609
Giovanna Melideo, Marco Mechelli, Roberto Baldoni, Alberto Marchetti Spaccamela

Topic 10
Programming Languages, Models, and Methods **617**
Paul H.J. Kelly, Sergei Gorlatch, Scott Baden, Vladimir Getov

HPF vs. SAC - A Case Study *(Research Note)* 620
Clemens Grelck, Sven-Bodo Scholz

Developing a Communication Intensive Application on the EARTH Multithreaded Architecture *(Distinguished Paper)* 625
Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, Guang R. Gao

On the Predictive Quality of BSP-like Cost Functions for NOWs 638
Mauro Bianco, Geppino Pucci

Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs 647
Siegfried Benkner, Thomas Brandes

The Skel-BSP Global Optimizer: Enhancing Performance Portability in Parallel Programming 658
Andrea Zavanella

A Theoretical Framework of Data Parallelism and Its Operational Semantics 668
Philippe Gerner, Eric Violard

A Pattern Language for Parallel Application Programs *(Research Note)* . 678
Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders

Oblivious BSP *(Research Note)* 682
Jesus A. Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José L. Roda, Casiano Rodriguez, Francisco de Sande

A Software Architecture for HPC Grid Applications *(Research Note)* . . . 686
Steven Newhouse, Anthony Mayer, John Darlington

Satin: Efficient Parallel Divide-and-Conquer in Java 690
Rob V. van Nieuwpoort, Thilo Kielmann, Henri E. Bal

Implementing Declarative Concurrency in Java 700
Rafael Ramirez, Andrew E. Santosa, Lee Wei Hong

Building Distributed Applications Using Multiple, Heterogeneous Environments 709
Paul A. Gray, Vaidy S. Sunderam

A Multiprotocol Communication Support for the Global Address Space Programming Model on the IBM SP 718
Jarek Nieplocha, Jialin Ju, Tjerk P. Straatsma

A Comparison of Concurrent Programming and Cooperative Multithreading 729
Takashi Ishihara, Tiejun Li, Eugene F. Fodor, Ronald A. Olsson

The Multi-architecture Performance of the Parallel Functional Language GpH *(Research Note)* 739
Philip W. Trinder, Hans-Wolfgang Loidl, Ed Barry Jr., M. Kei Davis, Kevin Hammond, Ulrike Klusik, Simon L. Peyton Jones, Álvaro J. Rebón Portillo

Novel Models for Or-Parallel Logic Programs: A Performance Analysis . 744
Vítor Santos Costa, Ricardo Rocha, Fernando Silva

Executable Specification Language for Parallel Symbolic Computation *(Research Note)* 754
Alexander B. Godlevsky, Ladislav Hluchý

Efficient Parallelisation of Recursive Problems Using Constructive Recursion *(Research Note)* 758
Magne Haveraaen

Development of Parallel Algorithms in Data Field Haskell *(Research Note)* 762
Jonas Holmerin, Björn Lisper

The ParCeL-2 Programming Language *(Research Note)* 767
Paul-Jean Cagnard

Topic 11
Numerical Algorithms for Linear and Nonlinear Algebra 771
Ulrich Rüde, Hans-Joachim Bungartz

Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free 774
David S. Wise

An Efficient Parallel Linear Solver with a Cascadic Conjugate Gradient Method: Experience with Reality 784
Peter Gottschling, Wolfgang E. Nagel

A Fast Solver for Convection Diffusion Equations Based on Nested Dissection with Incomplete Elimination 795
Michael Bader, Christoph Zenger

Low Communication Parallel Multigrid 806
Marcus Mohr

Parallelizing an Unstructured Grid Generator with a Space-Filling Curve Approach 815
Jörn Behrens, Jens Zimmermann

Solving Discrete-Time Periodic Riccati Equations on a Cluster *(Research Note)* 824
Peter Benner, Rafael Mayo, Enrique S. Quintana-Ortí, Vicente Hernández

A Parallel Optimization Scheme for Parameter Estimation in Motor Vehicle Dynamics *(Research Note)* 829
Torsten Butz, Oskar von Stryk, Thieß-Magnus Wolter

Sliding-Window Compression on the Hypercube *(Research Note)* 835
Charalampos Konstantopoulos, Andreas Svolos, Christos Kaklamanis

A Parallel Implementation of a Potential Reduction Algorithm for Box-Constrained Quadratic Programming 839
Marco D'Apuzzo, Marina Marino, Panos M. Pardalos, Gerardo Toraldo

Topic 12
European Projects **849**
Roland Wismüller, Renato Campo

NEPHEW: Applying a Toolset for the Efficient Deployment of a Medical Image Application on SCI-Based Clusters 851
Wolfgang Karl, Martin Schulz, Martin Völk, Sibylle Ziegler

SEEDS : Airport Management Database System 861
Tomáš Hrúz, Martin Bečka, Antonello Pasquarelli

HIPERTRANS: High Performance Transport Network Modelling and Simulation *(Research Note)* 869
Stephen E. Ijaha, Stephen C. Winter, Nasser Kalantery

Topic 13
Routing and Communication in Interconnection Networks **875**
Jose Duato

Experimental Evaluation of Hot-Potato Routing Algorithms on 2-Dimensional Processor Arrays *(Research Note)* 877
Constantinos Bartzis, Ioannis Caragiannis, Christos Kaklamanis, Ioannis Vergados

Improving the Up*/Down* Routing Scheme for Networks of Workstations 882
José Carlos Sancho, Antonio Robles

Deadlock Avoidance for Wormhole Based Switches 890
Ingebjørg Theiss, Olav Lysne

An Analytical Model of Adaptive Wormhole Routing with Deadlock Recovery *(Research Note)* 900
Mohamed Ould-Khaoua, Ahmad Khonsari

Analysis of Pipelined Circuit Switching in Cube Networks *(Research Note)* 904
Geyong Min, Mohamed Ould-Khaoua

A New Reliability Model for Interconnection Networks 909
Vicente Chirivella, Rosa Alcover

A Bandwidth Latency Tradeoff for Broadcast and Reduction 918
Peter Sanders, Jop F. Sibeyn

Optimal Broadcasting in Even Tori with Dynamic Faults *(Research Note)* 927
Stefan Dobrev, Imrich Vrt'o

Broadcasting in All-Port Wormhole 3-D Meshes of Trees *(Research Note)* 931
Petr Salinger, Pavel Tvrdík

Probability-Based Fault-Tolerant Routing in Hypercubes *(Research Note)* 935
Jehad Al-Sadi, Khaled Day, Mohamed Ould-Khaoua

Topic 14
Instruction-Level Parallelism and Processor Architecture 939
Kemal Ebcioglu

On the Performance of Fetch Engines Running DSS Workloads 940
Carlos Navarro, Alex Ramírez, Josep-L. Larriba-Pey, Mateo Valero

Cost-Efficient Branch Target Buffers 950
Jan Hoogerbrugge

Two-Level Address Storage and Address Prediction *(Research Note)* 960
Enric Morancho, José María Llabería, Àngel Olivé

Hashed Addressed Caches for Embedded Pointer Based Codes
(Research Note) 965
Marian Stanca, Stamatis Vassiliadis, Sorin Cotofana, Henk Corporaal

BitValue Inference: Detecting and Exploiting Narrow Bitwidth
Computations 969
Mihai Budiu, Majd Sakr, Kip Walker, Seth C. Goldstein

General Matrix-Matrix Multiplication Using SIMD Features of the PIII
(Research Note) 980
Douglas Aberdeen, Jonathan Baxter

Redundant Arithmetic Optimizations *(Research Note)* 984
Thomas Y. Yéh, Hong Wang

The Decoupled-Style Prefetch Architecture *(Research Note)* 989
Kevin D. Rich, Matthew K. Farrens

Exploiting Java Bytecode Parallelism by Enhanced POC Folding Model *(Research Note)* 994
Lee-Ren Ton, Lung-Chung Chang, Chung-Ping Chung

Cache Remapping to Improve the Performance of Tiled Algorithms 998
Kristof E. Beyls, Erik H. D'Hollander

Code Partitioning in Decoupled Compilers 1008
Kevin D. Rich, Matthew K. Farrens

Limits and Graph Structure of Available Instruction-Level Parallelism *(Research Note)* 1018
Darko Stefanović, Margaret Martonosi

Pseudo-vectorizing Compiler for the SR8000 *(Research Note)* 1023
Hiroyasu Nishiyama, Keiko Motokawa, Ichiro Kyushima, Sumio Kikuchi

Topic 15
Object Oriented Architectures, Tools, and Applications 1029
Gul A. Agha

Debugging by Remote Reflection 1031
Ton Ngo, John Barton

Compiling Multithreaded Java Bytecode for Distributed Execution *(Distinguished Paper)* 1039
Gabriel Antoniu, Luc Bougé, Philip J. Hatcher, Mark MacBeth, Keith McGuigan, Raymond Namyst

A More Expressive Monitor for Concurrent Java Programming 1053
Hsin-Ta Chiao, Chi-Houng Wu, Shyan-Ming Yuan

An Object-Oriented Software Framework for Large-Scale Networked Virtual Environments 1061
Frédéric Dang Tran, Anne Gérodolle

TACO - Dynamic Distributed Collections with Templates and Topologies 1071
Jörg Nolte, Mitsuhisa Sato, Yutaka Ishikawa

Object-Oriented Message-Passing with TPO++ *(Research Note)* 1081
Tobias Grundmann, Marcus Ritt, Wolfgang Rosenstiel

Topic 17
Architectures and Algorithms for Multimedia Applications 1085
Manfred Schimmler

Design of Multi-dimensional DCT Array Processors for Video Applications 1086
Shietung Peng, Stanislav Sedukhin

Design of a Parallel Accelerator for Volume Rendering 1095
Bertil Schmidt

Automated Design of an ASIP for Image Processing Applications *(Research Note)* 1105
Henjo Schot, Henk Corporaal

A Distributed Storage System for a Video-on-Demand Server *(Research Note)* 1110
Alice Bonhomme, Loïc Prylli

Topic 18
Cluster Computing 1115
Rajkumar Buyya, Mark Baker, Daniel C. Hyde, Djamshid Tavangarian

Partition Cast - Modelling and Optimizing the Distribution of Large Data Sets in PC Clusters *(Distinguished Paper)* 1118
Felix Rauch, Christian Kurmann, Thomas M. Stricker

A New Home-Based Software DSM Protocol for SMP Clusters 1132
Weiwu Hu, Fuxin Zhang, Haiming Liu

Encouraging the Unexpected: Cluster Management for OS and Systems Research *(Research Note)* 1143
Ronan Cunniffe, Brian A. Coghlan

Flow Control in ServerNet^R Clusters 1148
Vladimir Shurbanov, Dimiter Avresky, Pankaj Mehra, William Watson

The WMPI Library Evolution: Experience with MPI Development for Windows Environments 1157
Hernâni Pedroso, João Gabriel Silva

Implementing Explicit and Implicit Coscheduling in a PVM Environment *(Research Note)* 1165
Francesc Solsona, Francesc Giné, Porfidio Hernández, Emilio Luque

A Jini-Based Prototype Metacomputing Framework *(Research Note)* . . 1171
Zoltan Juhasz, Laszlo Kesmarki

SKElib: Parallel Programming with Skeletons in C 1175
Marco Danelutto, Massimiliano Stigliani

Token-Based Read/Write-Locks for Distributed Mutual Exclusion 1185
Claus Wagner, Frank Mueller

On Solving a Problem in Algebraic Geometry by Cluster Computing *(Research Note)* 1196
Wolfgang Schreiner, Christian Mittermaier, Franz Winkler

PCI-DDC Application Programming Interface: Performance in User-Level Messaging *(Research Note)* 1201
Eric Renault, Pierre David, Paul Feautrier

A Clustering Approach for Improving Network Performance in Heterogeneous Systems *(Research Note)* 1206
Vicente Arnau, Juan M. Orduña, Salvador Moreno, Rodrigo Valero, Aurelio Ruiz

Topic 19
Metacomputing 1211
Alexander Reinefeld, Geoffrey Fox, Domenico Laforenza, Edward Seidel

Request Sequencing: Optimizing Communication for the Grid 1213
Dorian C. Arnold, Dieter Bachmann, Jack Dongarra

An Architectural Meta-application Model for Coarse Grained Metacomputing 1223
Stephan Kindermann, Torsten Fink

Javelin 2.0: Java-Based Parallel Computing on the Internet 1231
Michael O. Neary, Alan Phipps, Steven Richman, Peter Cappello

Data Distribution for Parallel CORBA Objects 1239
Tsunehiko Kamachi, Thierry Priol, Christophe René

Topic 20
Parallel I/O and Storage Technology 1251
Rajeev Thakur, Rolf Hempel, Elizabeth Shriver, Peter Brezany

Towards a High-Performance Implementation of MPI-IO on Top of GPFS 1253
Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice E. Koniges, Alison White

Design and Evaluation of a Compiler-Directed Collective I/O Technique 1263
Gokhan Memik, Mahmut T. Kandemir, Alok Choudhary

Effective File-I/O Bandwidth Benchmark 1273
Rolf Rabenseifner, Alice E. Koniges

Instant Image: Transitive and Cyclical Snapshots in Distributed Storage Volumes 1284
Prasenjit Sarkar

Scheduling Queries for Tape-Resident Data 1292
Sachin More, Alok Choudhary

Logging RAID – An Approach to Fast, Reliable, and Low-Cost Disk Arrays 1302
Ying Chen, Windsor W. Hsu, Honesty C. Young

Topic 21

Problem Solving Environments **1313**
José C. Cunha, David W. Walker, Thierry Priol, Wolfgang Gentzsch

AMANDA - A Distributed System for Aircraft Design 1315
Hans-Peter Kersken, Andreas Schreiber, Martin Strietzel, Michael Faden, Regine Ahrem, Peter Post, Klaus Wolf, Armin Beckert, Thomas Gerholt, Ralf Heinrich, Edmund Kügeler

Problem Solving Environments: Extending the Rôle of Visualization Systems 1323
Helen Wright, Ken Brodlie, Jason Wood, Jim Procter

An Architecture for Web-Based Interaction and Steering of Adaptive Parallel/Distributed Applications 1332
Rajeev Muralidhar, Samian Kaur, Manish Parashar

Computational Steering in Problem Solving Environments *(Research Note)* 1340
David Lancaster, Jeff S. Reeve

Implementing Problem Solving Environments for Computational Science *(Research Note)* 1345
Omer F. Rana, Maozhen Li, Matthew S. Shields, David W. Walker, David Golby

Vendor Session

Pseudovectorization, SMP, and Message Passing on the Hitachi SR8000-F1 1351
Matthias Brehm, Reinhold Bader, Helmut Heller, Ralf Ebner

Index of Authors 1363

Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations

David E. Keyes

Department of Mathematics & Statistics
Old Dominion University, Norfolk VA 23529-0077, USA,
Institute for Scientific Computing Research
Lawrence Livermore National Laboratory, Livermore, CA 94551-9989 USA
Institute for Computer Applications in Science & Engineering
NASA Langley Research Center, Hampton, VA 23681-2199, USA
keyes@icase.edu,
http://www.math.odu.edu/∼keyes

Abstract. Simulations of PDE-based systems, such as flight vehicles, the global climate, petroleum reservoirs, semiconductor devices, and nuclear weapons, typically perform an order of magnitude or more below other scientific simulations (e.g., from chemistry and physics) with dense linear algebra or N-body kernels at their core. In this presentation, we briefly review the algorithmic structure of typical PDE solvers that is responsible for this situation and consider possible architectural and algorithmic sources for performance improvement. Some of these improvements are also applicable to other types of simulations, but we examine their consequences for PDEs: potential to exploit orders of magnitude more processor-memory units, better organization of the simulation for today's and likely near-future hierarchical memories, alternative formulations of the discrete systems to be solved, and new horizons in adaptivity. Each category is motivated by recent experiences in computational aerodynamics at the 1 Teraflop/s scale.

1 Introduction

While certain linear algebra and computational chemistry problems whose computational work requirements are superlinear in memory requirements have executed at 1 Teraflop/s, simulations of PDE-based systems remain "mired" in the hundreds of Gigaflop/s on the same machines. A review the algorithmic structure of typical PDE solvers that is responsible for this situation suggests possible avenues for performance improvement towards the achievement of the remaining four orders of magnitude required to reach 1 Petaflop/s.

An ideal 1 Teraflop/s computer of today would be characterized by approximately 1,000 processors of 1 Gflop/s each. (However, due to inefficiencies within the processors, a machine sustaining 1 Teraflop/s of useful computation is more practically characterized as about 4,000 processors of 250 Mflop/s each.) There

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1–17, 2000.
© Springer-Verlag Berlin Heidelberg 2000

are two extreme pathways by which to reach 1 Petaflop/s from here: 1,000,000 processors of 1 Gflop/s each (only wider), or 10,000 processors of 100 Gflop/s each (mainly deeper). From the point of view of PDE simulations on Eulerian grids, either should suit.

We begin in §2 with a brief and anecdotal review of progress in high-end computational PDE solution and a characterization of the computational structure and complexity of grid-based PDE algorithms.

A simple bulk-synchronous scaling argument (§3) suggests that continued expansion of the number of processors is feasible as long as the architecture provides a global reduction operation whose time-complexity is sublinear in the number of processors. However, the cost-effectiveness of this brute-force approach towards petaflop/s is highly sensitive to frequency and latency of global reduction operations, and to modest departures from perfect load balance.

Looking internal to a processor (§4), we argue that there are only two intermediate levels of the memory hierarchy that are essential to a typical domain-decomposed PDE simulation, and therefore that most of the system cost and performance cost for maintaining a deep multilevel memory hierarchy could be better invested in improving access to the relevant workingsets, associated with individual local stencils (matrix rows) and entire subdomains. Improvement of local memory bandwidth and multithreading — together with intelligent prefetching, perhaps through processors in memory to exploit it — could contribute approximately an order of magnitude of performance within a processor relative to present architectures. Sparse problems will never have the locality advantages of dense problems, but it is only necessary to stream data at the rate at which the processor can consume it, and what sparse problems lack in locality, they can make up for by scheduling. With statically discretized PDEs, the access patterns are persistent. The usual ramping up of processor clock rates and the width or multiplicity of instructions issued are other obvious avenues for per-processor computational rate improvement, but only if memory bandwidth is raised proportionally.

Besides these two classes of architectural improvements — more and better-suited processor/memory elements — we consider two classes of algorithmic improvements: some that improve the raw flop rate and some that increase the scientific value of what can be squeezed out of the average flop.

In the first category (§5), we mention higher-order discretization schemes, especially of discontinuous or mortar type, orderings that improve data locality, and iterative methods that are less synchronous than today's.

It can be argued that the second category of algorithmic improvements does not belong in a discussion focused on computational rates, at all. However, since the ultimate purpose of computing is insight, not petaflop/s, it must be mentioned as part of a balanced program, especially since it is not conveniently orthogonal to the other approaches. We therefore include a brief pitch (§6) for revolutionary improvements in the practical use of problem-driven algorithmic adaptivity in PDE solvers — not just better system software support for well understood discretization-error driven adaptivity, but true polyalgorithmic and

multiple-model adaptivity. To plan for a "bee-line" port of existing PDE solvers to petaflop/s architectures and to ignore the demands of the next generation of solvers will lead to petaflop/s platforms whose effectiveness in scientific and engineering computing might be scientifically equivalent only to less powerful but more versatile platforms. The danger of such a pyrrhic victory is real.

Each of the four "sources" of performance improvement mentioned above to aid in advancing from current hundreds of Gflop/s to 1 Pflop/s is illustrated with precursory examples from computational aerodynamics. Such codes have been executed in the hundreds of Gflop/s on up to 6144 processors of the ASCI Red machine of Intel and also on smaller partitions of the ASCI Blue machines of IBM and SGI and large Cray T3Es. (Machines of these architecture families comprise 7 of the Top 10 and 63 of the Top 100 installed machines worldwide, as of June 2000 [3].) Aerodynamics codes share in the challenges of other successfully parallelized PDE applications, though not comprehensive of all difficulties. They have also been used to compare numerous uniprocessors and examine vertical aspects of the memory system. Computational aerodynamics is therefore proposed as typical of workloads (nonlinear, unstructured, multicomponent, multiscale, etc.) that ultimately motivate the engineering side of high-end computing.

Our purpose is not to argue for specific algorithms or programming models, much less specific codes, but to identify algorithm/architecture stresspoints and to provide a requirements target for designers of tomorrow's systems.

2 Background and Complexity of PDEs

Many of the "Grand Challenges" of computational science are formulated as PDEs (possibly among alternative formulations). However, PDE simulations have frequently been absent in Bell Prize competitions (see Fig. 1). PDE simulations require a balance among architectural components that is not necessarily met in a machine designed to "max out" on traditional benchmarks. The justification for building petaflop/s architectures undoubtedly will (and should) include PDE applications. However, cost effective use of petaflop/s machines for PDEs requires further attention to architectural and algorithmic matters. In particular, a memory-centric, rather than operation-centric view of computation needs further promotion.

2.1 PDE Varieties and Complexities

The systems of PDEs that are important to high-end computation are of two main classifications: *evolution* (e.g., time hyperbolic, time parabolic) or *equilibrium* (e.g, elliptic, spatially hyperbolic or parabolic). These types can change type from region to region, or can be mixed in the sense of having subsystems of different types (e.g., parabolic with elliptic constraint, as in incompressible Navier-Stokes). They can be scalar or multicomponent, linear or nonlinear, but with all of the algorithmic accompanying variety, memory and work requirements after discretization can often be characterized in terms of five discrete parameters:

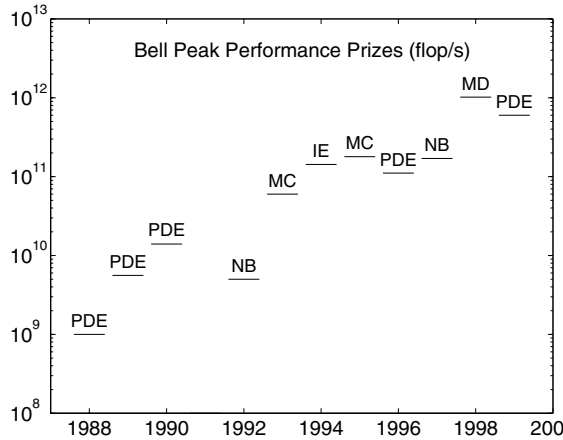


Fig. 1. Bell Prize Peak Performance computations for the decade spanning 1 Gflop/s to 1 Tflop/s. Legend: "PDE" – partial differential equations, "IE" integral equations, "NB" n-body problems, "MC" Monte Carlo problems, "MD" molecular dynamics. See Table 1 for further details.

Table 1. Bell Prize Peak Performance application and architecture summary. Prior to 1999, PDEs had successfully competed against other applications with more intensive data reuse only on special-purpose machines (vector or SIMD) in static, explicit formulations. ("NWT" is the Japanese Numerical Wind Tunnel.) In 1999, three of four finalists were PDE-based, all on SPMD hierarchical distributed memory machines.

Year	Type	Application	Gflop/s	System	No. procs.
1988	*PDE*	structures	1.0	*Cray Y-MP*	8
1989	*PDE*	seismic	5.6	*CM-2*	2,048
1990	*PDE*	seismic	14	*CM-2*	2,048
1992	NB	gravitation	5.4	Delta	512
1993	MC	Boltzmann	60	CM-5	1,024
1994	IE	structures	143	Paragon	1,904
1995	MC	QCD	179	NWT	128
1996	*PDE*	CFD	111	*NWT*	160
1997	NB	gravitation	170	ASCI Red	4,096
1998	MD	magnetism	1,020	T3E-1200	1,536
1999	*PDE*	CFD	627	*ASCI BluePac*	5,832

- N_x, number of spatial grid points (10^6–10^9)
- N_t, number of temporal grid points (1–unbounded)
- N_c, number of unknown components defined at each gridpoint (1–10^2)
- N_a, auxiliary storage per point (0–10^2)
- N_s, gridpoints in one conservation law "stencil" (5–50)

In these terms, memory requirements, M, are approximately $N_x \cdot (N_c + N_a + N_c^2 \cdot N_s)$. This is sufficient to store the entire physical state and allows workspace for an implicit Jacobian of the dense coupling of the unknowns that participate in the same stencil, but not enough for a full factorization of the same Jacobian. Computational work, W, is approximately $N_x \cdot N_t \cdot (N_c + N_a + N_c^2 \cdot (N_c + N_s))$. The last term represents updating of the unknowns and auxiliaries (equation-of-state and constitutive data, as well as temporarily stored fluxes) at each gridpoint on each timestep, as well as some preconditioner work on the sparse Jacobian of dense point-blocks.

From these two simple estimates comes a basic resource scaling "law" for PDEs. For equilibrium problems, in which solution values are prescribed on the boundary and interior values are adjusted to satisfy conservation laws in each cell-sized control volume, the work scales with the number of cells times the number of iteration steps. For optimal algorithms, the iteration count is constant independent of the fineness of the spatial discretization, but for commonplace and marginally "reasonable" implicit methods, the number of iteration steps is proportional to resolution in single spatial dimension. An intuitive way to appreciate this is that in pointwise exchanges of conserved quantities, it requires as many steps as there are points along the minimal path of mesh edges for the boundary values to be felt in the deepest interior, or for errors in the interior to be swept to the boundary, where they are absorbed. (Multilevel methods, when effectively designed, propagate these numerical signals on all spatial scales at once.) For evolutionary problems, work scales with the number of cells or vertices size times the number of time steps. CFL-type arguments place latter on order of resolution in single spatial dimension. In either case, for 3D problems, the iteration or time dimension is like an extra power of a single the spatial dimension, so Work $\propto$ (Memory)$^{4/3}$, with N_c, N_a, and N_s regarded as fixed. The proportionality constant can be adjusted over a very wide range by both discretization (high-order implies more work per point and per memory transfer) and by algorithmic tuning. This is in contrast to the classical Amdahl-Case Rule, that would have work and memory directly proportional. It is architecturally significant, since it implies that a petaflop/s-class machine can be somewhat "thin" on total memory, which is otherwise the most expensive part of the machine. However, memory bandwidth is still at a premium, as discussed later. In architectural practice, memory and processing power are usually increased in fixed proportion, by adding given processor-memory elements. Due to this discrepency between the linear and superlinear growth of work with memory, it is not trivial to design a single processor-memory unit for a wide range of problem sizes.

If frequent time frames are to be captured, other resources — disk capacity and I/O rates — must both scale linearly with W, more stringently than for memory.

2.2 Typical PDE Tasks

A typical PDE solver spends most of its time apart from pre- and post-processing and I/O in four phases, which are described here in the language of a vertex-centered code:

- Edge-based "stencil op" loops (resp., dual edge-based if cell-centered), such as residual evaluation, approximate Jacobian evaluation, and Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
- Vertex-based loops (resp., cell-based, if cell-centered), such as state vector and auxiliary vector updates
- Sparse, narrow-band recurrences, including approximate factorization and back substitution
- Global reductions: vector inner products and norms, including orthogonalization/conjugation and convergence progress and stability checks

The edge-based loops require near-neighbor exchanges of data for the construction of fluxes. They reuse data from memory better than the vertex-based and sparse recurrence stages, but today they are typically limited by a shortage of load/store units in the processor relative to arithmetic units and cache-available operands. The edge-based loop is key to performance optimization, since in a code that is not dominated by linear algebra this is where the largest amount of time is spent, and also since it contains a vast excess of instruction-level concurrency (or "slackness").

The vertex-based loops and sparse recurrences are purely local in parallel implementations, and therefore free of interprocessor communication. However, they typically stress the memory bandwidth within a processor/memory system the most, and are typically limited by memory bandwidth in their execution rates.

The global reductions are the bane of scalability, since they require some type of all-to-all communication. However, their communication and arithmetic volume is extremely low. The vast majority of flops go into the first three phases listed.

The insight that edge-based loops are load/store-limited, vertex-based loops and recurrences memory bandwith-limited, and reductions communication-limited is key to understanding and improving the performance of PDE codes. The effect of an individual "fix" may not be seen in most of the code until after an unrelated obstacle is removed.

2.3 Concurrency, Communication, and Synchronization

Explicit PDE solvers have the generic form:

$$\mathbf{u}^{\ell} = \mathbf{u}^{\ell-1} - \Delta t^{\ell} \cdot \mathbf{f}(\mathbf{u}^{\ell-1}),$$

where $\mathbf{u}^\ell$ is the vector of unknowns at time level ℓ, $\mathbf{f}$ is the flux function, and Δt^ℓ is the ℓth timestep. Let a domain of N discrete data be partitioned over an ensemble of P processors, with N/P data per processor. Since ℓ-level quantities appear only on the left-hand side, concurrency is pointwise, $\mathcal{O}(N)$. The communication-to-computation ratio has surface-to-volume scaling: $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. The range of communication for an explicit code is nearest-neighbor, except for stability checks in global time-step computation. The computation is bulk-synchronous, with synchronization frequency once per time-step, or $\mathcal{O}\left((\frac{N}{P})^{-1}\right)$. Observe that both communication-to-computation ratio and communication frequency are constant in a scaling of fixed memory per node. However, if P is increased with fixed N, they rise. Storage per point is low, compared to an implicit method. Load balance is straightforward for static quasi-uniform grids with uniform physics. Grid adaptivity or spatial nonuniformities in the cost to evaluate $\mathbf{f}$ make load balance potentially nontrivial. Adaptive load-balancing is a crucial issue in much (though not all) real-world computing, and its complexity is beyond this review. However, when a computational grid and its partitions are quasi-steady, the analysis of adaptive load-balancing can be usefully decoupled from the analysis of the rest of the solution algorithm.

Domain-decomposed implicit PDE solvers have the form:

$$\frac{\mathbf{u}^l}{\Delta t^l} + \mathbf{f}(\mathbf{u}^l) = \frac{\mathbf{u}^{l-1}}{\Delta t^l}, \quad \Delta t^l \to \infty.$$

Concurrency is pointwise, $\mathcal{O}(N)$, except in the algebraic recurrence phase, where it is only subdomainwise, $\mathcal{O}(P)$. The communication-to-computation ratio is still *mainly* surface-to-volume, $\mathcal{O}\left((\frac{N}{P})^{-1/3}\right)$. Communication still *mainly* nearest-neighbor, but convergence checking, orthogonalization/conjugation steps, and hierarchically coarsened problems add nonlocal communication. The synchronization frequency is usually more than once per grid-sweep, up to the dimension of the Krylov subspace of the linear solver, since global conjugations need to be performed to build up the latter: $\mathcal{O}\left(K(\frac{N}{P})^{-1}\right)$. The storage per point is higher, by factor of $\mathcal{O}(K)$. Load balance issues are the same as for the explicit case.

The most important message from this section is that a large variety of practically important PDEs can be characterized rather simply in terms of memory and operation complexity and relative distribution of communication and computational work. These simplifications are directly related to quasi-static grid-based data structures and the spatially and temporally uniform way in which the vast majority of points interior to a subdomain are handled.

3 Source #1: Expanded Number of Processors

As popularized in [5], Amdahl's law can be defeated if serial (or bounded concurrency) sections make up a nonincreasing fraction of total work as problem size and processor count scale together. This is the case for most explicit or iterative implicit PDE solvers parallelized by decomposition into subdomains. Simple, back-of-envelope parallel complexity analyses show that processors can

be increased as rapidly, or almost as rapidly, as problem size, assuming load is perfectly balanced. There is, however, an important caveat that tempers the use of large Beowulf-type clusters: the processor network must also be scalable. Of course, this applies to the protocols, as well as to hardware. In fact, the entire remaining four orders of magnitude to get to 1 Pflop/s could be met by hardware expansion alone. However, it is important to remember that this does not mean that fixed-size applications of today would run 10^4 times faster; this argument is based on memory-problem size scaling.

Though given elsewhere [7], a back-of-the-envelope scalability demonstration for bulk-synchronized PDE stencil computations is sufficiently simple and compelling to repeat here. The crucial observation is that both explicit and implicit PDE solvers periodically cycle between compute-intensive and communicate-intensive phases, making up one macro iteration.

Given complexity estimates of the leading terms of the concurrent computation (per iteration phase), the concurrent communication, and the synchronization frequency; and a model of the architecture including the internode communication (network topology and protocol reflecting horizontal memory structure) and the on-node computation (effective performance parameters including vertical memory structure); one can formulate optimal concurrency and optimal execution time estimates, on per-iteration basis or overall (by taking into account any granularity-dependent convergence rate).

For three-dimensional simulation computation costs (per iteration) assume and idealized cubical domain with:

- n grid points in each direction, for total work $N = \mathcal{O}(n^3)$
- p processors in each direction, for total processors $P = \mathcal{O}(p^3)$
- execution time per iteration, An^3/p^3 (where A includes factors like number of components at each point, number of points in stencil, number of auxiliary arrays, amount of subdomain overlap)
- n/p grid points on a side of a single processor's subdomain
- neighbor communication per iteration (neglecting latency), Bn^2/p^2
- global reductions at a cost of $C \log p$ or $Cp^{1/d}$ (where C includes synchronization frequency as well as other topology-independent factors)
- A, B, C are all expressed in the same dimensionless units, for instance, multiples of the scalar floating point multiply-add.

For the tree-based global reductions with a logarithmic cost, we have a total wall-clock time per iteration of

$$T(n,p) = A\frac{n^3}{p^3} + B\frac{n^2}{p^2} + C \log p.$$

For optimal p, $\frac{\partial T}{\partial p} = 0$, or $-3A\frac{n^3}{p^4} - 2B\frac{n^2}{p^3} + \frac{C}{p} = 0$, or (with $\theta \equiv \frac{32 \cdot B^3}{243 \cdot A^2 C}$),

$$p_{opt} = \left(\frac{3A}{2C}\right)^{1/3} \left(\left[1 + (1 - \sqrt{\theta})\right]^{1/3} + \left[1 - (1 - \sqrt{\theta})\right]^{1/3}\right) \cdot n.$$

This implies that the number of processors along each dimension, p, can grow with n without any "speeddown" effect. The optimal running time is

$$T(n, p_{\text{opt}}(n)) = \frac{A}{\rho^3} + \frac{B}{\rho^2} + C\log(\rho n),$$

where $\rho = \left(\frac{3A}{2C}\right)^{1/3} \left(\left[1 + (1-\sqrt{\theta})\right]^{1/3} + \left[1 - (1-\sqrt{\theta})\right]^{1/3} \right)$. In limit of global reduction costs dominating nearest-neighbor costs, $B/C \to 0$, leading to

$$p_{opt} = (3A/C)^{1/3} \cdot n,$$

and

$$T(n, p_{\text{opt}}(n)) = C\left[\log n + \frac{1}{3}\log\frac{A}{C} + const.\right].$$

We observe the direct proportionality of execution time to synchronization cost times frequency, C. This analysis is on a per iteration basis; fuller analysis would multiply this cost by an iteration count estimate that generally depends upon n and p; see [7]. It shows that an arbitrary factor of performance can be gained by following processor number with increasing problem sizes. Many multiple-scale applications of high-end PDE simulation (e.g., direct Navier-Stokes at high Reynolds numbers) can absorb all conceivable boosts in discrete problem size thus made available, yielding more and more science along the way.

The analysis above is for a memory-scaled problem; however, even a fixed-size PDE problem can exhibit excellent scalability over reasonable ranges of P, as shown in Fig. 2 from [4].

4 Source #2: More Efficient Use of Faster Processors

Current low efficiencies of sparse codes can be improved if regularity of reference is exploited through memory-assist features. PDEs have a simple, periodic workingset structure that permits effective use of prefetch/dispatch directives. They also have lots of slackness (process concurrency in excess of hardware concurrency). Combined with processor-in-memory (PIM) technology for efficient memory gather/scatter to/from densely used cache-block transfers, and also with multithreading for latency that cannot be amortized by sufficiently large block transfers, PDEs can approach full utilization of processor cycles. However, high bandwidth is critical, since many PDE algorithms do only $\mathcal{O}(N)$ work for $\mathcal{O}(N)$ gridpoints' worth of loads and stores.

Through these technologies, one to two orders of magnitude can be gained by first catching up to today's clocks, and then by following the clocks into the few-GHz range.

4.1 PDE Workingsets

The workingset is a time-honored notion in the analysis of memory system performance [2]. Parallel PDE computations have a smallest, a largest, and a spectrum of intermediate workingsets. The smallest is the set of unknowns, geometry

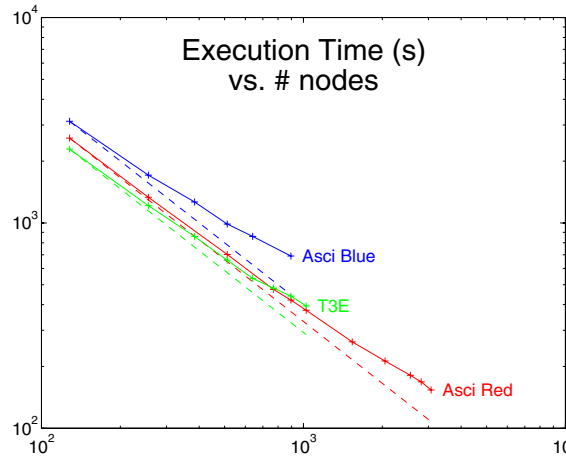


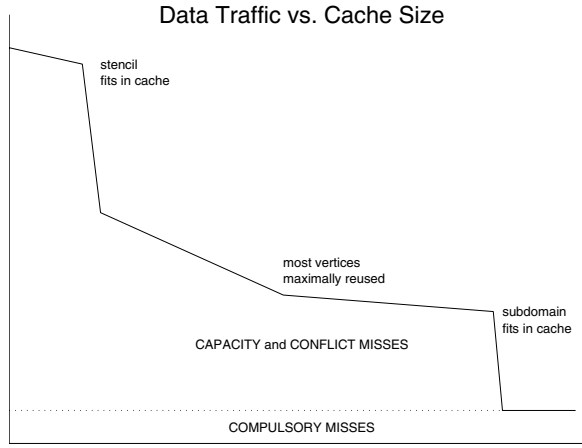
Fig. 2. Log-log plot of execution time vs. processor number for a full Newton-Krylov-Schwarz solution of an incompressible Euler code on two ASCI machines and a large T3E, up to at least 768 nodes each, and up to 3072 nodes of ASCI Red.

data, and coefficients at a multicomponent stencil. Its size is $N_s \cdot (N_c^2 + N_c + N_a)$ (relatively sharp). The largest is the set of unknowns, geometry data, and coefficients in an entire subdomain. Its size is $(N_x/P) \cdot (N_c^2 + N_c + N_a)$ (also relatively sharp) The intermediate workingsets are the data in neighborhood collections of gridpoints/cells that are reused within neighboring stencils.

As successive workingsets "drop" into a level of memory, capacity misses, (and with effort conflict misses) disappear, leaving only the one-time compulsory misses (see Fig. 3).

Architectural and coding strategies can be based on workingset structure. There is no performance value in any memory level with capacity larger than what is required to store all of the data associated with a subdomain. There is little performance value in memory levels smaller than the subdomain but larger than required to permit full reuse of most data within each subdomain subtraversal (middle knee, Fig. 3). After providing an L1 cache large enough for smallest workingset (associated with a stencil), and multiple independent stencils up to desired level of multithreading, all additional resources should be invested in a large L2 cache. The L2 cache should be of write-back type and its population under *user* control (e.g., prefetch/dispatch directives), since it is easy to determine when a data element is fully used within a given mesh sweep. Since this information has persistence across many sweeps it is worth determining and exploiting it. Tables describing grid connectivity are built (after each grid rebalancing) and stored in PIM. This meta-data is used to pack/unpack densely-used cache lines during subdomain traversal.

Fig. 3. Thought experiment for cache traffic for PDEs, as function of the size of the cache, from small (large traffic) to large (compulsory miss traffic only), showing knees corresponding to critical workingsets.



The left panel of Fig. 4 shows a set of twenty vertices in a 2D airfoil grid in the lower left portion of the domain whose working data are supposed to fit in cache simultaneously. In the right panel, the window has shifted in such a way that a majority of the points left behind (all but those along the upper boundary) are fully read (multiple times) and written (once) for this sweep. This is an unstructured analog of compiler "tiling" a regular multidimensional loop traversal. This corresponds to the knee marked "most vertices maximally reused" in Fig 3.

To illustrate the effects of reuse of a different type in a simple experiment that does not require changing cache sizes or monitoring memory traffic, we provide in Table 2 some data off three different machines for incompressible and compressible Euler simulation, from [6]. The unknowns in the compressible case are organized into 5×5 blocks, whereas those in the compressible case are organized into 4×4 blocks, by the nature of the physics. Data are intensively reused within a block, especially in the preconditioning phase of the algorithm. This boosts the overall performance by 7–10%.

The cost of greater per-processor efficiency arranged in these ways is the programming complexity of managing data traversals, the space to store gather/scatter tables in PIM, and the time to rebuild these tables when the mesh or physics changes dynamically.

Fig. 4. An unstructured analogy to the compiler optimization of "tiling" for a block of twenty vertices (*courtesy of D. Mavriplis*).

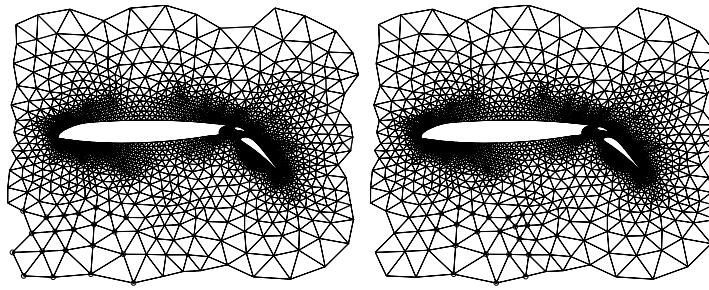


Table 2. Mflop/s per processor for highly L1- and register-optimized unstructured grid Euler flow code. Per-processor utilization is only 8% to 27% of peak. Slightly higher figure for compressible flow reflects larger number of components coupled densely at a gridpoint.

	Origin 2000		SP		T3E-900	
Processor	R10000		P2SC (4-card)		Alpha 21164	
Instr. Issue	2		4		2	
Clock (MHz)	250		120		450	
Peak Mflop/s	500		480		900	
Application	Incomp.	Comp.	Incomp.	Comp.	Incomp.	Comp.
Actual Mflop/s	126	137	117	124	75	82
Pct. of Peak	25.2	27.4	24.4	25.8	8.3	9.1

5 Source #3: More Architecture-Friendly Algorithms

Algorithmic practice needs to catch up to architectural demands. Several "one-time" gains remain that could improve data locality or reduce synchronization frequency, while maintaining required concurrency and slackness. "One-time" refers to improvements by small constant factors, nothing that scales in N or P. Complexities are already near their information-theoretic lower bounds for some problems, and we reject increases in flop rates that derive from *less* efficient algorithms, as defined by parallel execution time. These remaining algorithmic performance improvements may cost extra memory or they may exploit shortcuts of numerical stability that occasionally backfire, making performance modeling less predictable However, perhaps as much as an order of magnitude of performance remains here.

Raw performance improvements from algorithms include data structure reorderings that improve locality, such as interlacing of all related grid-based data structures and ordering gridpoints and grid edges for L1/L2 reuse. Dis-

cretizations that improve locality include such choices as higher-order methods (which lead to denser couplings between degrees of freedom than lower-order methods) and vertex-centering (which, for the same tetrahedral grid, leads to denser blockrows than cell-centering, since there are many more than four nearest neighbors). Temporal reorderings that improve locality include block vector algorithms (these reuse cached matrix blocks; the vectors in the block are independent) and multi-step vector algorithms (these reuse cached vector blocks; the vectors have sequential dependence).

Temporal reorderings may also reduce the synchronization penalty but usually at a threat to stability. Synchronization *frequency* may be reduced by deferred orthogonalization and pivoting and speculative step selection. Synchronization *range* may be reduced by replacing a tightly coupled global process (e.g., Newton) with loosely coupled sets of tightly coupled local processes (e.g., Schwarz).

Precision reductions make bandwidth seem larger. Lower precision representation in memory of preconditioner matrix coefficients or other poorly known data causes no harm to algorithmic convergence rates, as along as the data are expanded to full precision before arithmetic is done on them, after they are in the CPU.

As an illustration of the effects of spatial reordering, we show in Table 3 from [4] the Mflop/s per processor for processors in five families based on three versions of an unstructured Euler code: the original F77 vector version, a version that has been interlaced so that all data defined at a gridpoint is stored near-contiguously, and after a vertex reordering designed to maximally reuse edge-based data. Reordering yields a factor of 2.6 (Pentium II) up to 7.5 (P2SC) on this the unstructured grid Euler flow code.

Table 3. Improvements from spatial reordering: uniprocessor Mflop/s, with and without optimizations.

Processor	clock	Interlacing, Edge Reord.	Interlacing (only)	Original
R10000	250	126	74	26
P2SC (2-card)	120	97	43	13
Alpha 21164	600	91	44	33
Pentium II (Linux)	400	84	48	32
Pentium II (NT)	400	78	57	30
Ultra II	300	75	42	18
Alpha 21164	450	75	38	14
604e	332	66	34	15
Pentium Pro	200	42	31	16

6 Source #4: Algorithms Delivering More "Science per Flop"

Some algorithmic improvements do not improve flop rate, but lead to the same scientific end in reduced time or at lower hardware cost. They achieve this by requiring less memory and fewer operations than other methods, usually through some form of adaptivity. Such adaptive programs are more complicated and less thread-uniform than those they improve upon in quality/cost ratio. It is desirable that petaflop/s machines be "general purpose" enough to run the "best" algorithms. This is not daunting conceptually, but it puts an enormous premium on dynamic load balancing. An order of magnitude or more in execution time can be gained here for many problems.

Adaptivity in PDEs is usually through of in terms of spatial discretization. Discretization type and order are varied to attain the required accuracy in approximating the continuum everywhere without over-resolving in smooth, easily approximated regions. Fidelity-based adaptivity changes the continuous formulation to accommodate required phenomena everywhere without enriching in regions where nothing happens. A classical aerodynamics example is a full potential model in the farfield coupled to a boundary layer near no-slip surfaces. Stiffness-based adaptivity changes the solution algorithm to provide more powerful, robust techniques in regions of space-time where the discrete problem is linearly or nonlinearly "stiff," without extra work in nonstiff, locally well-conditioned regions

Metrics and procedures for effective adaptivity strategies are well developed for some discretization techniques, such as method-of-lines to stiff initial-boundary value problems in ordinary differential equations and differential algebraic systems and finite element analysis for elliptic boundary value problems. It is fairly wide open otherwise. Multi-model methods are used in *ad hoc* ways in numerous commercially important engineering codes, e.g., Boeing's TRANAIR code [8]. Polyalgorithmic solvers have been demonstrated in principle, but rarely in the "hostile" environment of high-performance multiprocessing. Sophisticated software approaches (e.g., object-oriented programming) make advanced adaptivity easier to manage. Advanced adaptivity may require management of hierarchical levels of synchronization — within a region or between regions. User-specification of hierarchical priorities of different threads may be required — so that critical-path computations can be given priority, while subordinate computations fill unpredictable idle cycles with other subsequently useful work.

To illustrate the opportunity for localized algorithmic adaptivity, consider the steady-state shock simulation described in Fig. 5 from [1]. During the period between iterations 3 and 15 when the shock is moving slowly into position, only problems in local subdomains near the shock need be solved to high accuracy. To bring the entire domain into adjustment by solving large, ill-conditioned linear algebra problems for every minor movement of the shock on each Newton iteration is wasteful of resources. An algorithm that adapts to nonlinear stiffness would seek to converge the shock location before solving the rest of the subdomain with high resolution or high algebraic accuracy.

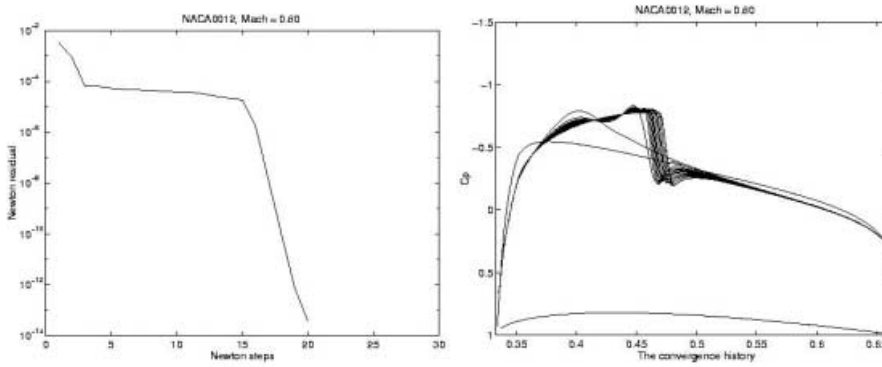


Fig. 5. Transonic full potential flow over NACA airfoil, showing (left) residual norm at each of 20 Newton iterations, with a plateau between iterations 3 and 15, and (right) shock developing and creeping down wing until "locking" into location at iteration 15, while the rest of flow field is "held hostage" to this slowly converging local feature.

7 Summary of Performance Improvements

We conclude by summarizing the types of performance improvements that we have described and illustrated on problems that can be solved on today's tera-scale computers or smaller. In reverse order, together with the possible performance factors available, they are:

- Algorithms that deliver more "science per flop"
  - possibly large problem-dependent factor, through adaptivity (but we won't count this towards rate improvement)
- Algorithmic variants that are more architecture-friendly
  - expect **half** an order of magnitude, through improved locality and relaxed synchronization
- More efficient use of processor cycles, and faster processor/memory
  - expect **one-and-a-half** orders of magnitude, through memory-assist language features, PIM, and multithreading
- Expanded number of processors
  - expect **two** orders of magnitude, through dynamic balancing and extreme care in implementation

Extreme concurrency — one hundred thousand heavyweight processes and a further factor of lightweight threads — are necessary to fully exploit this aggressive agenda. We therefore emphatically mention that PDEs do not arise individually from within a vacuum. Computational engineering is not about individual large-scale analyses, done fast and well-resolved and "thrown over the wall." Both results of an analysis and their sensitivities are desired. Often multiple operation points for a system to be simulated are known *a priori*, rather

than sequentially. The sensitivities may be fed back into an optimization process constrained by the PDE analysis. Full PDE analyses may also be inner iterations in a multidisciplinary computation. In such contexts, "petaflop/s" may mean 1,000 analyses running somewhat asynchronously with respect to each other, each at 1 Tflop/s. This is clearly a less daunting challenge and one that has better synchronization properties for exploiting such resources as "The Grid" than one analysis running at 1 Pflop/s.

As is historically the case, the high end of scientific computing will drive technology improvements across the entire information technology spectrum. This is ultimately the most compelling reason for pushing on through the next four orders of magnitude.

Acknowledgements

The author would like to thank his direct collaborators on computational examples reproduced in this chapter from earlier published work: Kyle Anderson, Satish Balay, Xiao-Chuan Cai, Bill Gropp, Dinesh Kaushik, Lois McInnes, and Barry Smith. Ideas and inspiration for various in various sections of this article have come from discussions with Shahid Bokhari, Rob Falgout, Paul Fischer, Kyle Gallivan, Liz Jessup, Dimitri Mavriplis, Alex Pothen, John Salmon, Linda Stals, Bob Voigt, David Young, and Paul Woodward. Computer resources have been provided by DOE (Argonne, Lawrence Livermore, NERSC, and Sandia), and SGI-Cray.

References

1. X.-C. Cai, W. D. Gropp, D. E. Keyes, R. G. Melvin and D. P. Young, Parallel Newton-Krylov-Schwarz Algorithms for the Transonic Full Potential Equation, SIAM J. Scientific Computing, **19**:246–265, 1998.
2. P. Denning, The Working Set Model for Program Behavior, Commun. of the ACM, **11**:323–333, 1968.
3. J. Dongarra, H.-W. Meuer, and E. Stohmaier, Top 500 Supercomputer Sites, `http://www.netlib.org/benchmark/top500.html`, June 2000.
4. W. D. Gropp, D. K. Kaushik, D. E. Keyes and B. F. Smith, Achieving High Sustained Performance in on Unstructured Mesh CFD Application, Proc. of Supercomputing'99 (CD-ROM), IEEE, Los Alamitos, 1999.
5. J. L. Gustafson, Re-evaluating Amdahl's Law, Commun. of the ACM **31**:532–533, 1988.
6. D. K. Kaushik, D. E. Keyes and B. F. Smith, NKS Methods for Compressible and Incompressible Flows on Unstructured Grids, Proc. of the 11th Intl. Conf. on Domain Decomposition Methods, C.-H. Lai, et al., eds., pp. 501–508, Domain Decomposition Press, Bergen, 1999.
7. D. E. Keyes, How Scalable is Domain Decomposition in Practice?, Proc. of the 11th Intl. Conf. on Domain Decomposition Methods, C.-H. Lai, et al., eds., pp. 282–293, Domain Decomposition Press, Bergen, 1999.

8. D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant and J. E. Bussoletti, A Locally Refined Rectangular Grid Finite Element Method: Application to Computational Fluid Dynamics and Computational Physics, J. Computational Physics **92**:1–66, 1991.

E2K Technology and Implementation

Boris Babayan

Elbrus International, Moscow, Russia
Boris.Babayan@elbrus.ru

For many years Elbrus team has been involved in the design and delivery of many generations of the most powerful Soviet computers. It has developed computers based on superscalar, shared memory multiprocessing and EPIC architectures. The main goal has always been to create a computer architecture which is fast, compatible, reliable and secure.
Main technical achievements of Elbrus Line computers designed by the team are: high speed, full compatibility, trustworthiness (program security, hardware fault tolerance), low power consumption and dissipation, low cost.

Elbrus-1 (1979): a superscalar RISC processor with out-of-order execution, speculative execution and register renaming. Capability-based security with dynamic type checking. Ten-CPU shared memory multiprocessor.
Elbrus-2 (1984): a ten-processor supercomputer
Elbrus-3 (1991): an EPIC based VLIW CPU. Sixteen-processor shared memory multiprocessor.

Our Approach is ExpLicit Basic Resource Utilization Scheduling - ELBRUS.

Elbrus Instruction Structure
Elbrus instructions fully and explicitly control all hardware resources for the compiler to perform static scheduling. Thus, Elbrus instruction is a variable size wide instruction consisting of one mandatory header syllable and up to 15 optional instruction syllables, each controlling a specific resource.

Advantages of ELBRUS Architecture

- Performance:
  The highest speed with given computational resources
  - Excellent cost performance
  - Excellent performance for the given level of memory subsystem
  - Well-defined set of compiler optimization needed to reach the limit
  - Highly universal
  - Can better utilize a big number of transistors in future chips
  - Better suited for high clock frequency implementation
- Simplicity:
  - More simple control logic
  - More simple and effective compiler optimization (explicit HW)
  - Easier and more reliable testing and HW correctness proof

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 18–21, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Elbrus approach allows most efficient designing of main data path resources (execution units, internal memories and interconnections without limitations from analysis and scheduling hardware).

Support of Straight-Line Program

- Wide instruction
- Variable size instruction (decreased code fetch throughput)
- Scoreboarding
- Multiport register file (split RF)
- Unified register file for integer and floating point units
- Increased number of registers in a single procedure context window with variable size
- Three independent register files for:
  - integer and FP data, memory address pointers
  - Boolean predicates
- HW implemented spill/fill mechanism (in a separate hidden stack)
- L1 Cache splitting

Support of Conditional Execution

- Exclusion of control transfer from data dependency graph. No need to conditionally control transfer for implementation of conditional expression semantics.
- Speculative execution explicitly program controlled
- Hoisting LOADs and operations across the basic blocks
- Predicated execution
- A big number of Boolean predicates and corresponding operations (in parallel with arithmetic ops)
- Elimination of output dependencies
- Introduction of control transfer statements during optimization
- Preparation to branch operations
- Icache preload
- Removing control transfer condition from critical path (unzipping)
- Short pipeline - fast branch
- Programmable branch predictor

Loops Support

- Loop overlapping
- Basing register references
- Basing predicate register references
- Support of memory access of array elements (automatic reference pointer forwarding)
- Array prefetch buffer
- Loop unroll support
- Loop control
- Recurrent loop support ("shift register")

Circuit Design
Advanced circuit design has been developed in Elbrus project to support extremely high clock frequency implementation. It introduces two new basic logic elements (besides traditional ones):

- universal self-reset logic with the following outstanding features
  - No losses for latches
  - No losses for clock skew
  - Time borrowing
  - Low power dissipation
- differential logic for high speed long distance signal transfer

This logic supports 25-30% better clock frequency compared to existing most advanced microprocessors.

Hardware Support of Binary Translation
Platform independent features:

- Two virtual spaces
- TLB design
  - Write protection
  - (self-modifying code)
  - I/O pages access
  - Protection
- Call/return cache
- Precise interrupt implementation (register context)

X86 platform specific features:

- Integer arithmetic and logical primitives
- Floating point arithmetic
- Memory access (including memory models support)
- LOCK prefix
- Peripheral support

E2K Ensures Intel Compatibility Including:

- Invisibility of the binary compiled code for original Intel code
- Run time code modifications
  - Run time code creation
  - Self-modifying code
  - Code modification in MP system by other CPUs
  - Code modification by external sources (PCI, etc.)
  - Modification of executables in code file
- Dynamic control transfer
- Optimizations of memory access order
- Proper interrupts handling
  - asynchronous
  - synchronous

Security
Elbrus security technology solves a critical problem of today - network security and full protection from viruses in Internet. Besides, it provides a perfect condition for efficient debugging and facilitates advanced technology for system programming.

Basic principle of security is extremely simple: "You should not steal". For information technology it implies that one should access only the data which one has created himself or which has been given to him from outside with certain access rights.

All data are accessed through address information (references, pointers). If pointers are handled properly, the above said is valid and the system is secure. Unfortunately, it is impossible to check statically pointers handling correctness without imposing undue restrictions on programming. For full, strong and efficient dynamic control of explicit pointer handling with no restrictions on programming HW support is required. And this is what Elbrus implements.

Traditional Approaches
To avoid pointer check problems, Java just throws away explicit pointer handling. This makes the language non-universal and still does not exclude dynamic check (for array ranges). C and C++ include explicit pointer handling but for efficiency reasons exclude dynamic check totally, which results in insecure programming.
Analysis of traditional approach:

1. Memory:
   Languages have pointer types, but they are presented by regular integer that can be explicitly handled by a user. No check of proper pointer handling – no security in memory.
2. File System:
   No pointer to a file data type. File reference is presented by a regular string. For the downloaded program to execute this reference, the file system root is made accessible to it. No protection in file system – good condition for virus reproduction.

Our Approach
Elbrus hardware supports dynamic pointer checking. For this reason each pointer is marked with special type bits. This does not lead to the use of non-standard DIMMs. In this way perfect memory protection and debugging facility are ensured. Using this technology we can run C and C++ in a fully secure mode. And Java becomes much more efficient.

File System and Network Security
To use these ideas in file system and Internet area C and C++ need to be extended by introduction of special data types - file or directory references. Now we can pass file references to the downloaded program. No need to provide access to the file system root for the downloaded program. Full security is ensured.

E2K is fast, compatible, reliable and secure. It is a real Internet oriented microprocessor.

Grid-Based Asynchronous Migration of Execution Context in Java Virtual Machines

Gregor von Laszewski[1], Kazuyuki Shudo[2], and Yoichi Muraoka[2]

[1] Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL, U.S.A.
gregor@mcs.anl.gov
[2] School of Science and Engineering, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan
{shudoh,muraoka}@muraoka.info.waseda.ac.jp

Abstract. Previous research efforts for building thread migration systems have concentrated on the development of frameworks dealing with a small local environment controlled by a single user. Computational Grids provide the opportunity to utilize a large-scale environment controlled over different organizational boundaries. Using this class of large-scale computational resources as part of a thread migration system provides a significant challenge previously not addressed by this community. In this paper we present a framework that integrates Grid services to enhance the functionality of a thread migration system. To accommodate future Grid services, the design of the framework is both flexible and extensible. Currently, our thread migration system contains Grid services for authentication, registration, lookup, and automatic software installation. In the context of distributed applications executed on a Grid-based infrastructure, the asynchronous migration of an execution context can help solve problems such as remote execution, load balancing, and the development of mobile agents. Our prototype is based on the migration of Java threads, allowing asynchronous and heterogeneous migration of the execution context of the running code.

1 Introduction

Emerging national-scale *Computational Grid* infrastructures are deploying advanced services beyond those taken for granted in today's Internet, for example, authentication, remote access to computers, resource management, and directory services. The availability of these services represents both an opportunity and a challenge an opportunity because they enable access to remote resources in new ways, a challenge: because the developer of thread migration systems may need to address implementation issues or even modify existing systems designs. The scientific problem-solving infrastructure of the twenty-first century will support the coordinated use of numerous distributed heterogeneous components, including advanced networks, computers, storage devices, display devices, and scientific instruments. The term *The Grid* is often used to refer to this emerging infrastructure [5]. NASA's Information Power Grid and the NCSA Alliance's National Technology Grid are two contemporary projects prototyping Grid systems; both build on a range of technologies, including many provided by the Globus project. Globus is a metacomputing toolkit that provides basic services for security, job submission, information, and communication.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 22–34, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The availability of a national Grid provides the ability to exploit this infrastructure with the next generation of parallel programs. Such programs will include mobile code as an essential tool for allowing such access enabled through *mobile agents*. Mobile agents are programs that can migrate between hosts in a network (or Grid), in order to find places of their own choosing. An essential part for developing mobile agent systems is to save the state of the running program before it is transported to the new host, and restored, allowing the program to continue where it left off. Mobile-agent systems differ from process-migration systems in that the agents move when they choose, typically through a *go* statement, whereas in a process-migration system the system decides when and where to move the running process (typically to balance CPU load) [9]. In an Internet-based environment mobile agents provide an effective choice for many applications as outlined in [11]. Furthermore, this applies also to Grid-based applications. Advantages include improvements in latency and bandwidth of client-server applications and reduction in vulnerability to network disconnection. Although not all Grid applications will need mobile agents, many other applications will find mobile agents an effective implementation technique for all or part of their tasks. The migration system we introduce in this paper is able to support mobile agents as well as process-migration systems, making it an ideal candidate for applications using migration based on the application as well as system requirements.

The rest of the paper is structured as follows. In the first part we introduce the thread migration system MOBA. In the second part we describe the extensions that allow the thread migration system to be used in a Grid-based environment. In the third part we present initial performance results with the MOBA system. We conclude the paper with a summary of lessons learned and a look at future activities.

2 The Thread Migration System MOBA

This paper describes the development of a Grid-based thread migration system. We based our prototype system on the thread migration system MOBA, although many of the services needed to implement such a framework can be used by other implementations.

The name *MOBA* is derived from *MOB*ile *A*gents, since this system was initially applied to the context of mobile agents [17][22][14][15]. Nevertheless, MOBA can also be applied to other computer science–related problems such as the remote execution of jobs [4][8][3]. The advantages of MOBA are threefold:

1. **Support for asynchronous migration**. Thread migration can be carried out without the awareness of the running code. Thus, migration allows entities outside the migrating thread to initiate the migration. Examples for the use of asynchronous migration are global job schedulers that attempt to balance loads among machines. The program developer has the clear advantage that minimal changes to the original threaded code are necessary to include sophisticated migration strategies.
2. **Support for heterogeneous migration**. Thread migration in our system is allowed between MOBA processes executed on platforms with different operating systems. This feature makes it very attractive for use in a Grid-based environment, which is by nature built out of a large number of heterogeneous computing components.

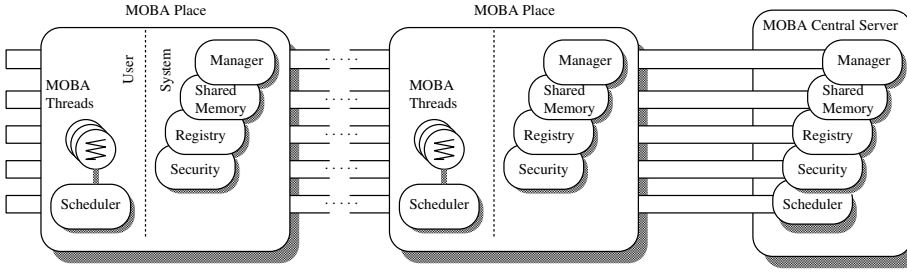


Fig. 1. The MOBA system components include MOBA places and a MOBA central server. Each component has a set of subcomponents that allow thread migration between MOBA places.

3. **Support for the execution of native code as part of the migrating thread.** While considering a thread migration system for Grid-based environments, it is advantageous to enable the execution of native code as part of the overall strategy to support a large and expensive code base, such as in scientific programming environments. MOBA will, in the near future, provide this capability. For more information on this subject we refer the interested reader to [17].

2.1 MOBA System Components

MOBA is based on a set of components that are illustrated in Figure 1. Next, we explain the functionality of the various components:

Place. Threads are created and executed in the *MOBA place* component. Here they receive external messages to move or decide on their own to move to a different place component. A MOBA place accesses a set of MOBA system components, such as manager, shared-memory, registry, and security. Each component has a unique functionality within the MOBA framework.

Manager. A single point of control is used to provide the control of startup and shutdown of the various component processes. The manager allows the user to get and set the environment for the respective processes.

Shared Memory: This component shares the data between threads.

Registry: The registry maintains necessary information — both static and dynamic — about all the MOBA components and the system resources. This information includes the OS name and version, installed software, machine attributes, and the load on the machines.

Security: The security component provides network-transparent programming interfaces for access control to all the MOBA components.

Scheduler: A MOBA place has access to user-defined components that handle the execution and scheduling of threads. The scheduling strategy can be provided through a custom policy developed by the user.

2.2 Programming Interface

We have designed the programming interface to MOBA on the principle of simplicity. One advantage in using MOBA is the availability of a user-friendly programming interface. For example, with only one statement, the programmer can instruct a thread to migrate; thus, only a few changes to the original code are necessary in order to augment an existent thread-based code to include thread migration. To enable movability of a thread, we instantiate a thread by using the `MobaThread` class instead of the normal Java `Thread` class. Specifically, the `MobaThread` class includes a method, called `goTo`, that allows the migration of a thread to another machine . In contrast to other mobile agent systems for Java [10][12][6], programmers using MOBA can enable thread migration with minor code modifications.

An important feature of MOBA is that migration can be ordered not only by the migrant but also by entities outside the migrant. Such entities include even threads that are running in the context of another user. In this case, the statement to migrate is included not in the migrant's code but in the thread that requests the move into its own execution context. To distinguish this action from the `goTo`, we have provided the method `moveTo`.

2.3 Implementation

MOBA is based on a specialized version of the Java Just-In-Time (JIT) interpreter. It is implemented as a plug-in to the Java Virtual Machine (JVM) provided by Sun Microsystems. Although MOBA is mostly written in Java, a small set of C functions enables efficient access to perform reflection and to obtain thread information such as the stack frames within the virtual machine. Currently, the system is supported on operating systems on which the Sun's JDK 1.1.x is ported. A port of MOBA based on JDK 1.2.x is currently under investigation. Our system allows heterogeneous migration [19] by handling the execution context in JVM rather than on a particular processor or in an operating system. Thus, threads in our system can migrate between JVMs on different platforms.

2.4 Organization of the Migration Facilities

To facilitate migration within our system, we designed MOBA as a layered architecture. The migration facilities of MOBA include introspection, object marshaling, thread externalization, and thread migration. Each of these facilities is supported and accessed through a library. The relationship and dependency of the migration facilities are depicted in Figure 2. The introspection library provides the same function as the reflection library that is part of the standard library of Java. Similarly, object marshaling provides the function of serialization, and thread externalization translates a state of the running thread to a byte stream.

The steps to translate a thread to a byte stream are summarized in Figure 3. In the first step, the attributes of the thread are translated. Such attributes include the name of the thread and thread priority. In the second step, all objects that are reachable from

the thread object are marshaled. Objects that are bound to file descriptors or other local resources are excluded from a migration. In the final step, the execution context is serialized. Since a context consists of contents of stack frames generated by a chain of method invocations, the externalizer follows the chain from older frames to newer ones and serializes the contents of the frames. A frame is located on the stack in a JVM and contains the state of a calling method. The state consists of a program counter, operands to the method, local variables, and elements on the stack, each of which is serialized in machine-independent form.

Together the facilities for externalizing threads and performing thread migration enabled us to design the components necessary for the MOBA system and to enhance the JIT compiler in order to allow asynchronous migration.

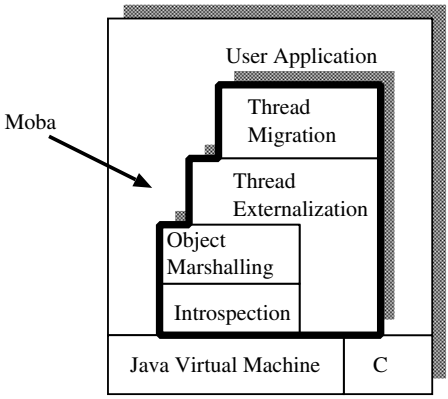


Fig. 2. Organization of MOBA thread migration facilities and their dependencies.

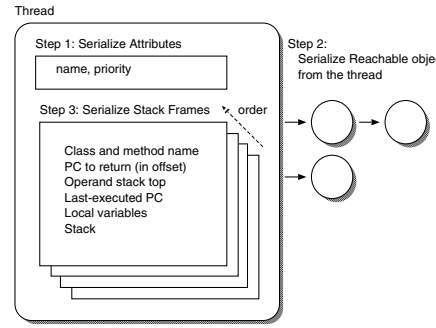


Fig. 3. Procedure to externalize a thread in MOBA.

2.5 Design Issues of Thread Migration in JVMs

In designing our thread migration system, we faced several challenges. Here we focus on five.

Nonpreemptive Scheduling. In order to enable the migration of the execution context, the migratory thread must be suspended at a *migration safe point*. Such migration safe points are defined within the execution of the JVM whenever it is in a consistent state. Furthermore, asynchronous migration within the MOBA system requires nonpreemptive scheduling of Java threads to prevent threads from being suspended at a not-safe point. Depending on the underlying (preemptive or nonpreemptive) thread scheduling system used in the JVM, MOBA supports either asynchronous or cooperative migration (that is, the migratory thread determines itself the destination). The availability of green threads will allow us to provide asynchronous migration.

Native Code Support. Most JVMs have a JIT runtime compiler that translates bytecode to the processors native code at runtime. To enable heterogeneous migration, a machine-independent representation of execution context is required. Unfortunately, most existing JIT compilers do not preserve a program counter on bytecode which is needed to reach a migration safe point. Only the program counter of the native code execution can be obtained by an existing JIT compiler. Fortunately, Sun's HotSpot VM [18] allows the execution context on bytecode to be captured during the execution of the generated native code since capturing the program counter on bytecode is also used for its dynamic deoptimization.

We are developing an enhanced version of the JIT compiler that checks, during the execution of native code, a flag indicating whether the request for capturing the context can be performed. This polling may have some cost in terms of performance, but we expect any decrease in performance to be small.

Selective Migration. In the most primitive migration system all objects reachable from the thread object are marshaled and replicated on the destination of the migration. This approach may cause problems related to limitations occuring during the access of system resources as documented in [17]. Selective migration may be able to overcome these problems, but the implementation is challenging because we must develop an algorithm determining the objects to be transferred. Additionally, the migration system must cooperate with a distributed object system, enabling remote reference and remote operation. Specifically, since the migrated thread must allow access to the remaining objects within the distributed object system, it must be tightly integrated within the JVM. It must allow the interchange of a local references and a remote references to support remote array access, field access, transparent replacement of a local object with a remote object, and so forth. Since no distributed object system implemented in Java (for example, Java RMI, Voyager, HORB, and many implementations of CORBA) satisfies these requirements, we have developed a distributed object system supported by the JIT compiler shuJIT [16] to provide these capabilities.

Marshaling Objects Tied to the Local Resource. A common problem in object migration systems is how to maintain objects that have some relation to resources specific to, say, a machine. Since MOBA does not allow to access objects that reside in a remote machine directly, it must copy or migrate the objects to the MOBA place issuing the request. Objects that depend on local resources (such a file and socket descriptors) are not moved within MOBA, but remain at the original fixed location [8][13].

Types of Values on the JVM Stack. In order to migrate an object from one machine to another, it is important to determine the type of the local object variables. Unfortunately, Sun's JVM does not provide a type stack operating in parallel to the value stack, such as the Sumatra interpreter [1]. Local variables and operands of the called method stay on the stack. The values may be 32-bit or 64-bit immediate values or references to objects. It is difficult to distinguish the types only by their values.

With a JVM like Sun's, we have either to infer the type from the value or to determine the type by a data flow analysis that traces the bytecode of the method (like a bytecode verifier). Since tracing bytecode to determine types is computationally expensive,

we developed a version of MOBA that infers the type from the value. Nevertheless, we recently determined that this capability is not sufficient to obtain a perfect inference and validation method. Thus, we are developing a modified JIT compiler that will provide stack frame maps [2] as part of Sun's ResearchVM.

3 Moba/G Service Requirements

The thread migration system MOBA introduced in the preceding sections is used as a basis for a Grid-enhanced version which we will call MOBA/G. Before we describe the MOBA/G system in more detail, we describe a simple Grid-enhanced scenario to outline our intentions for a Grid-based MOBA framework. First, we have to determine a subset of compute resources on which our MOBA system can be executed. To do so, we query the Globus Metacomputing Directory Service (MDS) while looking for compute resources on which Globus and the appropriate Java VM versions are installed and on which we have an account. Once we have identified a subset of all the machines returned by this query for the execution of the MOBA system, we transfer the necessary code base to the machine (if it is not already installed there). Then we start the MOBA places and register each MOBA place within the MDS. The communication between the MOBA places is performed in a secure fashion so that only the application user can decrypt the messages exchanged between them. A load-balancing algorithm is plugged into the running MOBA system that allows us to execute our thread-based program rapidly in the dynamically maintained MOBA places. During the execution of our program we detect that a MOBA place is not responding. Since we have designed our program with check-pointing, we are able to start new MOBA places on underutilized resources and to restart the failed threads on them. Our MOBA application finishes and deregisters from the Grid environment.

To derive such a version, we have tried to ask ourselves several questions:

1. What existent Grid services can be used by MOBA to enhance is functionality?
2. What new Grid services are needed to provide a Grid-based MOBA system?
3. Are any technological or implementation issues preventing the integration?

To answer the first two questions, we identified that the following services will be needed to enhance the functionality of MOBA in a Grid-based environment:

Resource Location and Monitoring Services. A resource location service is used to determine possible compute nodes on which a MOBA place can be executed. A monitoring service is used to observe the state and status of the Grid environment to help in scheduling the threads in the Grid environment. A combination of Globus services can be used to implement them.

Authentication and Authorization Service. The existent security component in MOBA is based on a simple centralized maintenance based on user accounts and user groups known in a typical UNIX system. This security component is not strong enough to support the increased security requirements in a Grid-based environment. The Globus project, however, provides a sophisticated security infrastructure that

can be used by MOBA. Authentication can be achieved with the concept of public keys. This security infrastructure can be used to augment many of the MOBA components, such as shared memory and the scheduler.

Installation and Execution Service. Once a computational resource has been discovered, an installation service is used to install a MOBA place on it and to start the MOBA services. This is a significant enhancement to the original MOBA architecture as it allows the shift from a static to a dynamic pool of resources. Our intention is to extend a component in the Globus toolkit to meet the special needs of MOBA.

Secure Communication Service. Objects in MOBA are exchanged over the IIOP protocol. One possibility is to use commercial enhancements for the secure exchange of messages between different places. Another solution is to integrate the Globus security infrastructure. The Globus project has initiated an independent project investigating the development of a CORBA framework using a security enhanced version of IIOP.

The services above can be based on a set of existing Grid services provided by the Globus project (compare Table 1). For the integration of MOBA and Globus we need consider only those services and components that increase the functionality of MOBA within a Grid-based environment.

Table 1. The Globus services that are used to build the MOBA/G thread migration system within a Grid-based environment. Services that are not available in the initial MOBA system are indcated with •.

MOBA/G Service	Service	Globus Component
MOBA Place startup	Resource Management	GRAM
MOBA Object migration	Communication	GlobusIO
• Secure Communication, Authentication, Secure component startup	Security	GSI
MOBA registry	Information	MDS
• Monitoring	Health and Status	HBM, NWS
• Remote Installation, Data Replication	Remote Data Access	GASS

Before we explain in more detail the integration of each of the services into the MOBA system, we point out that many of the services are accessible in Java through the Java CoG Kit. The Java CoG Kit [20][21] not only allows access to the Globus services, but also provides the benefit of using the Java framework as the programming model. Thus, it is possible to cast the services as JavaBenas and to use the sophisticated event and thread models as used in the programs to support the MOBA/G implementation. The relationship between Globus, the Java CoG Kit, and MOBA/G is based on a layered architecture as depicted in Figure 4.

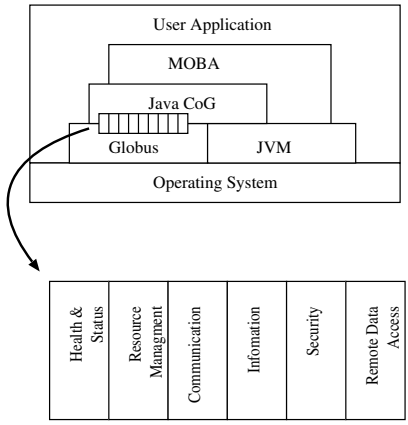


Fig. 4. The layered architecture of MOBA/G. The Java CoG Kit is used to access the various Globus Services.

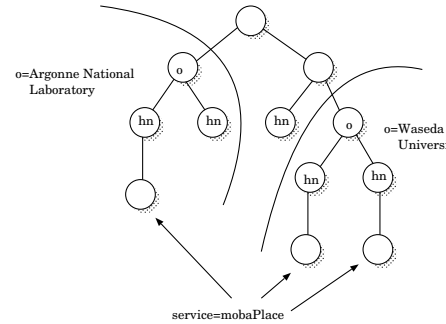


Fig. 5. The organizational directory tree of a distributed MOBA/G system between two organizations using three compute resources (hn) for running MOBA places.

3.1 Grid-Based Registration Service

One of the problems a Grid-based application faces is to identify the resources on which the application is executed. The Metacomputing Directory Service enables Grid application developers and users to register their services with the MDS. The Grid-based information service could be used in several ways:

1. The existing MOBA central registry could register its existence within the MDS. Thus all MOBA services would still interact with the original MOBA service. The advantage of including the MOBA registry within the MDS is that multiple MOBA places could be started with multiple MOBA registries, and each of the places could easily locate the necessary information from the MDS in order to set up the communication with the appropriate MOBA registry.

2. The information that is usually contained within the MOBA registry could be stored as LDAP objects within the distributed MDS. Thus, the functionality of the original MOBA registry could be replaced with a distributed registry based on the MDS functionality.

3. The strategies introduced in (1) and (2) could be mixed while registering multiple enhanced MOBA registries. These enhanced registries would allow the exchange of information between each other and thus function in a distributed fashion.

Which of the methods introduced above is used depends on the application. Applications with high throughput demand but few MOBA places are sufficiently supported by the original MOBA registry. Applications that have a large number of MOBA places but do not have high demands on the throughput benefit from a total distributed registry in the MDS. Applications that fall between these classes benefit from a modified MOBA distributed registry.

We emphasize that a distributed Grid-based information service must be able to deal in most cases with organizational boundaries (Figure 5). All of the MDS-based solutions discussed above provide this nontrivial ability.

3.2 Grid-Based Installation Service

In a Grid environment we foresee the following two possibilities for the installation of MOBA: (1) MOBA and Globus are already installed on the system, and hence we do not have to do anything; and (2) we have to identify a suitable machine on which MOBA can be installed. The following steps describe such an automatic installation process:

1. Retrieve a list of all machines that fulfill the installation requirements (e.g., Globus, JDK1.1, a particular OS-version, enough memory, accounts on which the user has access, platform-supported green-threads).
2. Select a subset of these machines on which to install MOBA.
3. Use a secure Grid-enabled ftp program to download MOBA in an appropriate installation space, and uncompress the distribution in this space.
4. Configure MOBA while using the provided auto-configure script, and complete the installation process.
5. Test the configuration, and, if successful, report and register the availability of MOBA on the machine.

3.3 Grid-Based Startup Service

Once MOBA is installed on a compute resource and a user decides to run a MOBA place on it, it has to be started together with all the other MOBA services to enable a MOBA system. The following steps are performed in order to do so:

1. Obtain the authentication through the Globus Security service to access the appropriate compute resource.
2. List all the machines on which a user can start a MOBA place.
3. For each compute resource in the list, start MOBA through the Java CoG interface to the Globus remote job startup service.

Depending on the way the registry service is run, additional steps may be needed to start it or to register an already running registry within the MDS.

3.4 Authentication and Authorization Service

In contrast to the existing MOBA security system, the Grid-based security service is far more sophisticated and flexible. It is based on GSI and allows integration with public keys as well as with Kerberos. First, the user must authenticate to the system. Using this Grid-based single-sign on security service allows the user to gain access to all the resources in the Grid without logging onto the various machines on the Grid environment on which the user has accounts, with potential different user names and passwords. Once authenticated, the user can submit remote job request that are executed with the appropriate security authorization for the remote machine. In this way a user can access remote files, create threads in a MOBA place, and initiate the migration of threads between MOBA places.

3.5 Secure Communication Service

The secure communication can be enabled while using the GlobusIO library and sending messages from one Globus machine to another. This service allows one to send any serializable object or simple message (e.g., thread migration, class file transfer, and commands to the MOBA command interpreter) to other MOBA places executed under Globus-enabled machines.

4 Conclusion

We have designed and implemented migration system for Java threads as a plug-in to an existing JVM that supports asynchronous migration of execution context. As part of this paper we discussed various issues, such as whether objects reachable from the migrant should be moved, how the types of values in the stack can be identified, how compatibility with JIT compilers can be achieved, and how system resources tied to moving objects should be handled. As a result of this analysis, we are designing a JIT compiler that improves our current prototype. It will support asynchronous and heterogeneous migration with execution of native code. The initial step to such a system is already achieved because we have already implemented a distributed object system based on the JIT compiler to support selective migration. Although this is an achievement by itself, we have enhanced our vision to include the emerging Grid infrastructure. Based on the availability of mature services provided as part of the Grid infrastructure, we have modified our design to include significant changes in the system architecture. Additionally, we have identified services that can be used by other Grid application developers. We feel that the integration of a thread migration system in a Grid-based environment has helped us to shape future activities in the Grid community, as well as to make improvements in the thread migration system.

Acknowledgments

This work was supported by the Research for the Future (RFTF) program launched by Japan Society for the Promotion of Science (JSPS) and funded by the Japanese government. The work performed by Gregor von Laszewski work was supported by the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. Globus research and development is supported by DARPA, DOE, and NSF.

References

1. Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*. Springer Verlag Lecture Notes in Computer Science, 1997.
2. Ole Agesen. GC points in a threaded environment. Technical Report SMLI TR-98-70, Sun Microsystems, Inc., December 1998. http://www.sun.com/research/jtech/pubs/.

3. Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transaction on Parallel and Distributed Systems*, 9(5):459–469, May 1998.
4. M. Raşit Eskicioğlu. Design Issues of Process Migration Facilities in Distributed System. *IEEE Technical Comittee on Operating Systems Newsletter*, 4(2):3–13, Winter 1989. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.
5. I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1998.
6. General Magic, Inc. Odyssey information. http://www.genmagic.com/technology/odyssey.html.
7. Satoshi Hirano. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications*, March 1997.
8. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transaction on Computer Systems*, 6(1):109–133, February 1988.
9. David Kotz and Robert S. Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, 33(3):7–13, August 1999.
10. Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, Inc., 1998.
11. Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.
12. ObjectSpace, Inc. Voyager. http://www.objectspace.com/products/Voyager/.
13. M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware mobile programs. In *Proceedings of USENIX 97*, January 1997.
14. Tatsuro Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Springer Lecture Notes in Computer Science for International Conference on Coordination Models and Languages(Coordination99)*, 1999.
15. Tatsurou Sekiguchi. JavaGo manual, 1998. http://web.yl.is.s.u-tokyo.ac.jp/amo/JavaGo/doc/.
16. Kazuyuki SHUDO. shuJIT—JIT compiler for Sun JVM/x86, 1998. http://www.shudo.net/jit/.
17. Kazuyuki Shudo and Yoichi Muraoka. Noncooperative Migration of Execution Context in Java Virtual Machines. In *Proc. of the First Annual Workshop on Java for High-Performance Computing (in conjunction with ACM ICS 99)*, Rhodes, Greece, June 1999.
18. Inc. Sun Microsystems. The Java HotSpot performance engine architecture. http://www.javasoft.com/products/hotspot/ whitepaper.html.
19. Marvin M. Theimer and Barry Hayes. Heterogeneous Process Migration by Recompilation. In *Proc. IEEE 11th International Conference on Distributed Computing Systems*, pages 18–25, 1991. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.
20. Gregor von Laszewski and Ian Foster. Grid Infrastructure to Support Science Portals for Large Scale Instruments. In *Proc. of the Workshop Distributed Computing on the Web (DCW)*, pages 1–16, Rostock, June 1999. University of Rostock, Germany.
21. Gregor von Laszewski, Ian Foster, Jarek Gawor, Warren Smith, and Steve Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM 2000 Java Grande Conference*, San Francisco, California, June 3-4 2000. http://www.extreme.indiana.edu/java00.
22. James E. White. *Telescript Technology: The Foundation of the Electronic Marketplace*. General Magic, Inc., 1994.

23. Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *The Second Conference on Object Ori ented Technology and Systems (COOTS) Proceedings*, pages 219–231, 1996.

Logical Instantaneity and Causal Order: Two "First Class" Communication Modes for Parallel Computing

Michel Raynal

IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
raynal@irisa.fr

Abstract. This paper focuses on two communication modes, namely *Logically Instantaneity* (LI) and *Causal Order* (CO). These communication modes address two different levels of quality of service in message delivery. LI means that it is possible to timestamp communication events with integers in such a way that (1) timestamps increase within each process and (2) the sending and the delivery events associated with each message have the same timestamp. So, there is a logical time frame in which for each message, the send event and the corresponding delivery events occur simultaneously. CO means that when a process delivers a message m, its delivery occurs in a context where the receiving process knows all the causal past of m. Actually, LI is a property strictly stronger than CO.
The paper explores these noteworthy communication modes. Their main interest lies in the fact that they deeply simplify the design of message-passing programs that are intended to run on distributed memory parallel machines or cluster of workstations.

Keywords: Causal Order, Cluster of Workstations, Communication Protocol, Distributed Memory, Distributed Systems, Logical Time, Logical Instantaneity, Rendezvous.

1 Introduction

Designing message-passing parallel programs for distributed memory parallel machines or clusters of workstations is not always a trivial task. In a lot of cases, it reveals to be a very challenging and error-prone task. That is why any system designed for such a context has to offer the user a set *Services* that simplify his programming task. The ultimate goal is to allow him to concentrate only on the problem he has to solve and not the technical details of the machine on which the program will run.

Among the services offered by such a system to upper layer application processes, *Communication Services* are of crucial importance. A communication service is defined by a pair of matching primitives, namely a primitive that allows to send a message to one or several destination processes and a primitive

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 35–42, 2000.
© Springer-Verlag Berlin Heidelberg 2000

that allows a destination process to receive a message sent to it. Several communication services can coexist within a system. A communication service is defined by a set of properties. From a user point of view, those properties actually define the *quality of service* (*QoS*) offered by the communication service to its users. These properties usually concern reliability and message ordering.

A *reliability* property states the conditions under which a message has to be delivered to its destination processes despite possible failures. An *ordering* property states the order in which messages have to be delivered; usually this order depends on the message sending order. FIFO, causal order (CO) [4,14] and *total order* (TO) [4] are the most encountered ordering properties [7]. Reliability and ordering properties can be combined to give rise to powerful communication primitives such as Atomic Broadcast [4] or Atomic Multicast to asynchronous groups

Another type of communication service is offered by CSP-like languages. This communication type assumes reliable processes and provides the so-called *rendezvous* (RDV) communication paradigm [2,8] (also called *synchronous* communication.) "A system has synchronous communications if no message can be sent along a channel before the receiver is ready to receive it. For an external observer, the transmission then looks like instantaneous and atomic. Sending and receiving a message correspond in fact to the same event" [5]. Basically, RDV combines synchronization and communication. From an operational point of view, this type of communication is called *blocking* because the sender process is blocked until the receiver process accepts and delivers the message. "While asynchronous communication is less prone to deadlocks and often allows a higher degree of parallelism (...) its implementation requires complex buffer management and control flow mechanisms. Furthermore, algorithms making use of asynchronous communication are often more difficult to develop and verify than algorithms working in a synchronous environment" [6]. This quotation expresses the relative advantages of synchronous communication with respect to asynchronous communication.

This paper focuses on two particular message ordering properties, namely, *Logical Instantaneity* (LI), and *Causal Order* (CO). The LI communication mode is weaker than RDV in the sense that it does not provide synchronization; more precisely, the sender of a message is not blocked until the destination processes are ready to deliver the message. But LI is stronger than CO (*Causally Ordered* communication). CO means that, if two sends are causally related [10] and concern the same destination process, then the corresponding messages are delivered in their sending order [4]. Basically, CO states that when a process delivers a message m, its delivery occurs in a context where the receiving process already knows the causal past of m. CO has received a great attention in the field of distributed systems, this is because it greatly simplifies the design of protocols solving consistency-related problems [14].

It has been shown that these communication modes form a strict hierarchy [6,15]. More precisely, RDV $\Rightarrow$ LI $\Rightarrow$ CO $\Rightarrow$ FIFO, where X $\Rightarrow$ Y means that if the communications satisfy the X property, they also satisfy the Y property. (More sophisticated communication modes can found in [1].) Of course, the less

constrained the communications are, the more efficient the corresponding executions can be. But, as indicated previously, a price has to be paid when using less constrained communications: application programs can be more difficult to design and prove, they can also require sophisticated buffer management protocols. Informally, LI provides the illusion that communications are done according to RDV, while actually they are done asynchronously. More precisely, LI ensures that there is a *logical* time frame with respect to which communications are synchronous.

This paper is mainly centered on the definition of the LI and CO communication modes. It is composed of four sections. Section 2 introduces the underlying system model. Then, Section 3 and Section 4 glance through the LI and CO communication modes, respectively. As a lot of literature has been devoted to CO, the paper content is essentially focused on LI.

2 Underlying System Model

2.1 Underlying Asynchronous Distributed System

The underlying asynchronous distributed system consists of a finite set P of n processes $\{P_1, \ldots, P_n\}$ that communicate and synchronize only by exchanging messages. We assume that each ordered pair of processes is connected by an asynchronous, reliable, directed logical channel whose transmission delays are unpredictable but finite[1]. The capacity of a channel is supposed to be infinite. Each process runs on a different processor, processors do not share a common memory, and there is no bound on their relative speeds.

A process can execute internal, send and receive operations. An internal operation does not involve communication. When P_i executes the operation $send(m, P_j)$ it puts the message m into the channel connecting P_i to P_j and continues its execution. When P_i executes the operation $receive(m)$, it remains blocked until at least one message directed to P_i has arrived, then a message is withdrawn from one of its input channels and delivered to P_i. Executions of internal, send and receive operations are modeled by internal, sending and receive events. Processes of a distributed computation are *sequential*; in other words, each process P_i produces a *sequence* of events $e_{i,1} \ldots e_{i,s} \ldots$ This sequence can be finite or infinite. Moreover, processes are assumed to be reliable.

Let H be the set of all the events produced by a distributed computation. This computation is modeled by the partially ordered set $\widehat{H} = (H, \xrightarrow{hb})$, where $\xrightarrow{hb}$ denotes the well-known Lamport's *happened-before* relation [10]. Let $e_{i,x}$ and $e_{j,y}$ be two different events:

$$e_{i,x} \xrightarrow{hb} e_{j,y} \Leftrightarrow \begin{cases} \quad i = j \wedge x < y \\ \vee\ \exists m : e_{i,x} = send(m, P_j) \wedge e_{j,y} = receive(m) \\ \vee\ \exists e : e_{i,x} \xrightarrow{hb} e \wedge e \xrightarrow{hb} e_{j,y} \end{cases}$$

[1] Note that channels are not required to be FIFO.

So, the underlying system model is the well known reliable asynchronous distributed system model.

2.2 Communication Primitives at the Application Level

The communication interface offered to application processes is composed of two primitives denoted SEND and DELIVER.

- The SEND$(m, dest_m)$ primitive allows a process to send a message m to a set of processes, namely $dest_m$. This set is defined by the sender process P_i (without loss of generality, we assume $P_i \not\in dest_m$). Moreover, every message m carries the identity of its sender: $m.sender = i$. The corresponding application level event is denoted SEND$_{m.sender}(m)$.
- The DELIVER(m) primitive allows a process (say P_j) to receive a message that has been sent to it by an other process (so, $P_j \in dest_m$). The corresponding application level event is denoted DELIVER$_j(m)$.

It is important to notice that the SEND primitive allows to multicast a message to a set of destination processes which is dynamically defined by the sending process.

3 Logically Instantaneous Communication

3.1 Definition

In the context of LI communication, when a process executes SEND$(m, dest_m)$ we say that it "LI-sends" m. When a process executes DELIVER(m) we say that it "LI-delivers" m. Communications of a computation satisfy the LI property if the four following properties are satisfied.

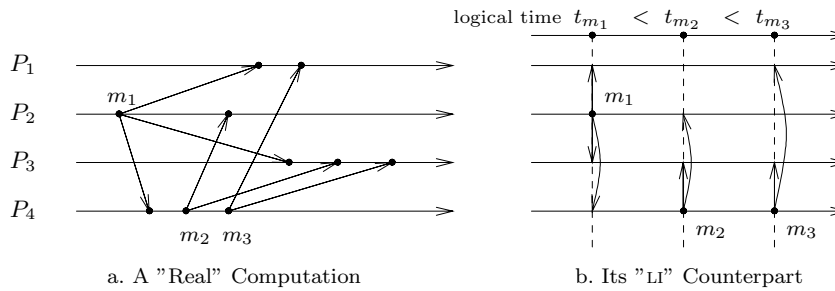
- *Termination*. If a process LI-sends m, then m is made available for LI-delivery at each process $P_j \in dest_m$. P_j effectively LI-delivers m when it executes the corresponding DELIVER primitive[2].
- *Integrity*. A process LI-delivers a message m at most once. Moreover, if P_j LI-delivers m, then $P_j \in dest_m$.
- *Validity*. If a process LI-delivers a message m, then m has been LI-sent by $m.sender$.
- *Logical Instantaneity*. Let $\mathbb{N}$ be the set of natural integers. This set constitutes the (logical) time domain. Let H_a be the set of all application level communication events of the computation. There exists a timestamping function $\mathcal{T}$ from H_a into $\mathbb{N}$ such that $\forall (e, f) \in H_a \times H_a$ [11]:

[2] Of course, for a message (that has been LI-sent) to be LI-delivered by a process $P_j \in dest_m$, it is necessary that P_j issues "enough" invocations of the DELIVER primitive. If m is the $(x+1)$th message that has to be LI-delivered to P_j, its LI-delivery at P_j can only occur if P_j has first LI-delivered the x previous messages and then invokes the DELIVER primitive.

(LI$_1$) e and f have been produced by the same process with e first
$$\Rightarrow \mathcal{T}(e) < \mathcal{T}(f)$$
(LI$_2$) $\forall m : \forall j \in dest_m : e = \text{SEND}_{m.sender}(m) \wedge f = \text{DELIVER}_j(m)$
$$\Rightarrow \mathcal{T}(e) = \mathcal{T}(f)$$

From the point of view of the communication of a message m, the event $\text{SEND}_{m.sender}(m)$ is the cause and the events $\text{DELIVER}_j(m)$ ($j \in dest_m$) are the effects. The termination property associates effects with a cause. The validity property associates a cause with each effect (in other words, there are no spurious messages). Given a cause, the integrity property specifies how many effects it can have and where they are produced (there are no duplicates and only destination processes may deliver a message). Finally, the logical instantaneity property specifies that there is a logical time domain in which the send and the deliveries events of every message occur at the same instant.

Figure 1.a describes communications of a computation in the usual space-time diagram. We have: $m_1.sender = 2$ and $dest_{m_1} = \{1, 3, 4\}$; $m_2.sender = m_3.sender = 4$, $dest_{m_2} = \{2, 3\}$ and $dest_{m_3} = \{1, 3\}$. These communications satisfy the LI property as shown by Figure 1.b. While RDV allows only the execution of Figure 1.b, LI allows more concurrent executions such as the one described by Figure 1.a.



a. A "Real" Computation b. Its "LI" Counterpart

Fig. 1. Logical Instantaneity

3.2 Communication Statements

In the context of LI communication, two types of statements in which communication primitives can be used by application processes are usually considered.

- *Deterministic Statement.* An application process may invoke the DELIVER primitive and wait until a message is delivered. In that case the invocation appears in a deterministic context (no alternative is offered to the process in case the corresponding SEND is not executed). In the same way, an application process may invoke the SEND primitive in a deterministic context.

– *Non-Deterministic Statement.* The invocation of a communication primitive in a deterministic context can favor deadlock occurrences (as it is the case, for example, when each process starts by invoking DELIVER.) In order to help applications prevent such deadlocks, we allow processes to invoke communication primitives in a non-deterministic statement (ADA and similar languages provide such non-deterministic statements). This statement has the following syntactical form:

$$\textbf{select\_com}\ \text{SEND}(m, dest_m)\ \textbf{or}\ \text{DELIVER}(m')\ \textbf{end\_select\_com}$$

This statement defines a non-deterministic context. The process waits until one of the primitives is executed. The statement is terminated as soon as a primitive is executed, a flag indicates which primitive has been executed. Actually, the choice is determined at runtime, according to the current state of communications.

3.3 Implementing LI Communication

Due to space limitation, there is not enough room to describe a protocol implementing the LI communication mode. The interested reader is referred to [12] where a very general and efficient protocol is presented. This protocol is based on a three-way handshake.

4 Causally Ordered Communication

4.1 Definition

In some sense Causal Order generalizes FIFO communication. More precisely, a computation satisfies the CO property if the following properties are satisfied:

– *Termination.* If a process CO-sends m, then m is made available for CO-delivery at each process $P_j \in dest_m$. P_j effectively CO-delivers m when it executes the corresponding DELIVER primitive.
– *Integrity.* A process CO-delivers a message m at most once. Moreover, if P_j CO-delivers m, then $P_j \in dest_m$.
– *Validity.* If a process CO-delivers a message m, then m has been CO-sent by $m.sender$.
– *Causal Order.* For any pair of message $m1$ and $m2$ such that CO-$send(m1) \xrightarrow{hb}$ CO-$send(m2)$ then, $\forall\ p_j \in dest_{m1} \cap dest_{m2}$, p_j CO-delivers $m1$ before $m2$.

Actually, CO constraints the non-determinism generated by the asynchrony of the underlying system. It forces messages deliveries to respect the causality order of their sendings.

Figure 2 depicts two distributed computation where messages are broadcast. Let us first look at the computation on the left side. We have $send(m1) \xrightarrow{hb} send(m2)$; moreover, m_1 and m_2 are deliverd in this order by each process. The sending of m_3 is causally related to neither m_1 nor m_2, hence no constraint

applies to its delivery. If follows that communication of this computation satisfies CO property. The reader can easily verify that the right computation does not satisfy the CO communication mode (the third process delivers m_1 after m_2, while their sendings are ordered the other way by $\stackrel{hb}{\rightarrow}$).

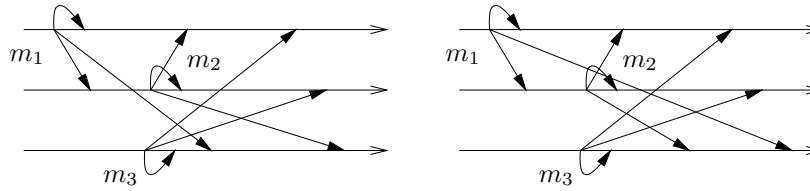


Fig. 2. Causal Order

4.2 Implementation Protocols

Basically, a protocol implementing causal order associates with each message a delivery condition. This condition depends on the current context of the receiving process (i.e., which messages it has already delivered) an on the context of the message (i.e., which message have been sent in the causal past of its sending).

The interested reader will find a basic protocol implementing causal order in [14]. More efficient protocols can be found in [3] for broadcast communication and in [13] for the general case (multicast to arbitrary subsets of processes). A formal (ad nice) study of protocols implementing the CO communication mode can be found in [9].

References

1. Ahuja M. and Raynal M., An implementation of Global Flush Primitives Using Counters. *Parallel Processing Letters*, Vol. 5(2):171-178, 1995.
2. Bagrodia R., Synchronization of Asynchronous Processes in CSP. *ACM TOPLAS*, 11(4):585-597, 1989.
3. Baldoni R., Prakash R., Raynal M. and Singhal M., Efficient Δ-Causal Broadcasting. *Journal of Computer Systems Science and Engineering*, 13(5):263-270, 1998.
4. Birman K.P. and Joseph T.A., Reliable Communication in the Presence of Failures. *ACM TOCS*, 5(1):47-76, 1987.
5. Bougé L., Repeated Snapshots in Distributed Systems with Synchronous Communications and their Implementation in CSP. *TCS*, 49:145-169, 1987.
6. Charron-Bost B., Mattern F. and Tel G., Synchronous, Asynchronous and Causally Ordered Communications. *Distributed Computing*, 9:173-191, 1996.
7. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press (S. Mullender Ed.), New-York, pp. 97-145, 1993.

8. Hoare C.A.R., Communicating Sequential Processes. *Communications of the ACM*, 21(8):666-677, 1978.
9. Kshemkalyani A.D. and Singhal M., Necessary and Sufficient Conditions on Information for Causal Message Ordering and their Optimal Implementation. *Distributed Computing*, 11:91-111, 1998.
10. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, 21(7):558-565, 1978.
11. Murty V.V. and Garg V.K., Synchronous Message Passing. *Tech. Report* TR ECE-PDS-93-01, University of Texas at Austin, 1993.
12. Mostefaoui A., Raynal M. and Verissimo P., Logically Instantaneous Communications in Asynchronous Distributed Systems. *5th Int. Conference on Parallel Computing Technologies (PACT'99)*, St-Petersburg, Springer Verlag LNCS 1662, pp. 258-270, 1999.
13. Prakash R., Raynal M. and Singhal M., An adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of Parallel and Distributed Computing*, 41:190-204, 1997.
14. Raynal M., Schiper A. and Toueg S., The Causal ordering Abstraction and a Simple Way to Implement it. *Information Processing Letters*, 39:343-351, 1991.
15. Soneoka T. and Ibaraki T., Logically Instantaneous Message Passing in Asynchronous Distributed Systems. *IEEE TC*, 43(5):513-527, 1994.

The TOP500 Project of the Universities Mannheim and Tennessee

Hans Werner Meuer

University of Mannheim, Computing Center
D-68131 Mannheim, Germany
Phone: +49 621 181 3176, Fax: +49 621 181 3178
meuer@rz.uni-mannheim.de
http://www.uni-mannheim.de/rum/members/meuer.html

Abstract

The TOP500 project was initiated in 1993 by Hans Meuer and Erich Strohmaier of the University of Mannheim. The first TOP500 list was published in cooperation with Jack Dongarra,University of Tennessee, at the 8th Supercomputer Conference in Mannheim. The TOP500 list has replaced the Mannheim Supercomputer Statistics published since 1986 and counting the worldwide installed vector systems.

After publishing the 15th TOP500 list of the most powerful computers worldwide in June 2000, we take stock:

In the beginning of the 1990s, while the MP vector systems reached their widest distribution, a new generation of MPP systems came on the market claiming to be able to substitute or even surpass the vector MPs. The increased competiveness of MPPs made it less and less meaningful to compile supercomputer statistics by just counting the vector computers. This was the major reason for starting the TOP500 project.

It appeares that the TOP500 list updated every 6 month since June 1993 is of major interest for the worldwide HPC community since it is a useful instrument to observe the HPC-market, to recognize market trends as well as those of architectures and technology.

The presentation will focus on the TOP500 project and our experience since seven years. The performance measure `Rmax` (best Linpack performance) will be discussed and in particular its limitations. Performace improvements over the last seven years are presented and will be compared with Moore's law. Projections based on the TOP500 data will be made in order to forecast e.g. the appearance of a Petaflop/s system.

The 15th list published in June on the occasion of the SC2000 Conference in Mannheim will be examined in some detail. Main emphasis will be on the trends mentioned before and visible through the continuation of the TOP500 list during the last 7 years. Especially the European situation will be addressed in detail. At the end of the talk our TOP500 Web site, `http://www.top500.org`, will be introduced.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 43–43, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Topic 01
Support Tools and Environments

Barton P. Miller and Michael Gerndt

Topic Chairmen

Parallelism is difficult, yet parallel programs are crucial to the high-performance needs of scientific and commercial applications. Success stories are plentiful; when parallelism works, the results are impressive. We see incredible results in such fields are computational fluid dynamics, quantum chromo dynamics, real-time animation, ab initio molecular simulations, climate modeling, macroeconomic forecasting, commodity analysis, and customer credit profiling.

But newcomers to parallel computing face a daunting task. Sequential thinking can often lead to unsatisfying parallel codes. Whether the task in to port an existing sequential code or to write a new code, There are the challenges of decomposing the problem in a suitable way, matching the structure of the computation to the architecture, and knowing the technical and stylistic tricks of a particular architecture needed to get good performance. Even experienced parallel programmers face a significant challenge when moving a program to a new architecture.

It is the job of the tool builder to somehow ease the task of designing, writing, debugging, tuning, and testing parallel programs. There have been notable successes in both the industrial and research world. But it is an continuing challenge. Our job, as tool builders, is made more difficult by a variety of factors:

1. Processors, architectures, and operating systems change faster than we can follow. Tool builders are constantly trying to improve their tools, but often are forced to spend too much time porting or adapting to new platforms.
2. Architectures are getting more complicated. The memory hierarchy continues to deepen, adding huge variability to access time. Processor designs now include aggressive out-of-order execution, providing the potential for great execution speed, but also penalizing poorly constructed code. As memories and processes get faster and more complicated, getting precise information about execution behaviors is more difficult.
3. Standards abound. There is the famous saying "The wonderful thing about standards is that there are so many of them!" It was only a short time ago that everyone was worried about various types of data-parallel Fortran; HPF was the magic language. PVM followed as a close second. Now, MPI is the leader, but does not standardized many of the important operations that tool-builders need to monitor. So, each vendor's MPI is a new challenge to support and the many independent MPI platforms (such as MPICH or LAM) present their own challenges. Add Open/MP to the mix, and life becomes much more interesting. And don't forget the still-popular Fortran dialects, such as Fortran 77 and Fortran 90.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 45–46, 2000.
© Springer-Verlag Berlin Heidelberg 2000

4. Given the wide variety of platforms, and fast rate of change, users want the same tools each time they move to a new platform. This movement increases the pressure to spend time porting tools (versus developing new ideas and tools). Heterogeneous and multi-mode parallelism only make the task more challenging.

The long-term ideal is for an environment and language where we can write our programs in a simple notation and have them automatically parallelized and tuned. Unfortunately, this is not likely to happen in the near future, so the demand for tool builders and their tools will remain high. This current instance of the Support Tools and Environments topic present new results in this area, hopefully bringing us closer to our eventual goal.

Visualization and Computational Steering in Heterogeneous Computing Environments

Sabine Rathmayer

Institut für Informatik, LRR
Technische Universität München,
D-80290 München, Deutschland
Sabine.Rathmayer@in.tum.de
http://wwwbode.in.tum.de

Abstract. Online visualization and computational steering has been an active research issue for quite a few years. The use of high performance scientific applications in heterogeneous computing environments still reveals many problems. With OViD we address problems that arise from distributed applications with no global data access and present additional collaboration features for distributed teams working with one simulation. OViD provides an interface for parallel and distributed scientific applications to send their distributed data to OViD's object pool. Another interface allows multiple visualization systems to attach to a running simulation, access the collected data from the pool, and send steering data back.

1 Introduction

During the last couple of years parallel computing has evolved from a purely scientific domain to industrial acceptance and use. At the same time, high performance computing platforms have moved from parallel machines to highly distributed systems. Highly distributed in the sense of different types of architectures - where architectures will include all from dedicated parallel machines to single processor personal computers - within the global Internet. Parallel and distributed scientific applications have been and are used within this context mostly in a batch-oriented manner. Visualization of results , steering (changing parameters) of the simulation and collaboration is done after the simulation run. It has long been recognized that users would benefit enormously if an interactive exploration of scientific simulations was possible.

Online interaction with scientific applications comprises online visualization, steering and collaboration. *Online visualization* aims at continuously providing the user with intermediate results of the simulations. This means that data for the visualization system has to be provided at the runtime of the program. The possibility of writing the results to files shall be excluded. There must be interaction between the application and the visualization system. In Fig. 1 we can see that some component is needed which is taking care of the communication and data management. Data from the parallel application processes has to be

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 47–56, 2000.
© Springer-Verlag Berlin Heidelberg 2000

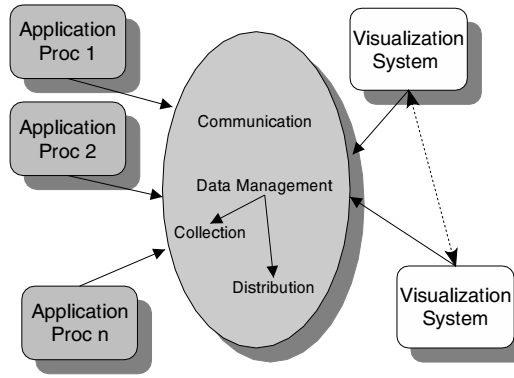


Fig. 1. Interaction diagram between visualization systems and parallel application

collected and distributed to any connected visualization system. If application and visualization were directly coupled the huge data transfers that come with high performance simulations would be a communication bottleneck.

interactive steering again requires interaction between the visualization system and the parallel application processes. Parameters or other steering information have to be sent to the different parallel processes according to the data distribution. Steering also requires synchronization among parallel processes. If there is more than one visualization system some token mechanism has to be implemented to guarantee correct steering actions among all participants.

Collaboration between distributed teams is another aspect of interaction in scientific computing. Users at different locations often want to discuss the behavior or the results of a running simulation. To provide collaboration for interactive simulations one can choose from different levels of difficulty. For many cases it might already be sufficient to enable consistent visualization and steering for distributed User Interfaces (UI). Additional personal communication would then be required via for example telephone. Which information needs to be shared among the different UIs and how it can be managed, depends on what the users actually want to be able to do with the software. Another look at Fig. 1 shows that we need some sort of data management pool to distribute the data to anyone who requests it.

The question now is how these features can be realized in todays heterogeneous computing environments. It is important that the features can be integrated into already existing applications and visualization systems with reasonable effort. The system must be platform-independent, extendible and efficient. We will show the features of our system OVID along with descriptions of its components and use.

2 Related Work

We look at some of the related work with respect to the above mentioned features as well as to the kinds of applications that are supported by the systems.

CUMULVS was developed in the Computer Science and Mathematics Division at *O*ak *R*idge *N*ational *L*aboratory [8]. It was designed to allow online visualization and steering of parallel PVM applications with an AVS viewer. An additional feature it provides is a fault-tolerance mechanism. It allows to checkpoint an application and then restart from the saved state instead of completely re-running from the start.

Applications are assumed to be iterative and contain regular data structures. These data structures are distributed in an HPF[1]-like manner. Applications have to be instrumented with library calls to define data for visualization, steering, and checkpointing, as well as their storage and distribution. At the end of each iteration the data transfer to the visualization system is initiated.

CUMULVS allows to attach multiple viewers to an application. Yet, since a synchronization between application and viewer is required for data exchange, this can heavily influence the performance of the application.

CSE is the abbreviation for *C*omputational *S*teering *E*nvironment and is developed at the Center for Mathematics and Computer Science in Amsterdam [1].

The CSE system is designed as a client/server model consisting of a server called the data manager and separate client processes, i.e satellites. The latter are either an application (which is packaged into a satellite) or different user interfaces. The data manager acts as a blackboard for the exchange of data values between the satellites. The satellites can create, open, close, read, and write variables in the data base. Therefore steering is implemented by explicit changing the values in the data manager. Special variables are used for synchronization of the satellites.

SCIRun is the developed by the *S*cientific *C*omputing and *I*maging group at the University of Utah. The system is designed as a workbench or so-called problem solving environment for the scientific user to create, debug, and use scientific simulations [2].

SCIRun was mainly designed for the development of new applications but also allows to incorporate existing applications. It is based on a data flow programming model with modules, ports, data types, and connections as its elements. The user can create a new steering application via a visual programming tool. Modules generate or update data types (as vectors or matrices) which are directed to the next module in the flow model via a port. Connections finally are used to connect different modules. If the user updates any of the parameters in a module the module is re-executed and all changes are automatically propagated to all downstream modules. There are predefined modules for monitoring the application and data visualization.

[1] High Performance Fortran

The system so far runs on machines that have a global address space. There is work in progress to also support distributed memory machines. No collaboration is possible among different users.

3 OViD

OViD has been developed according to the following design aspects. We provide an **open interface architecture** for different parallel and distributed applications as well as varying visualization systems. Regarding this, simplicity and extensibility are the main aspects. The system must run on **heterogeneous parallel and distributed computing environments**. **Multiple levels of synchronization** between parallel applications and visualization systems are needed. To detect for example errors in the simulation it might be necessary to force a tight coupling whereas in production runs a short runtime of the program is wanted. **Multiple visualization front-ends** must be able to attach to OViD. As we have shown before, it is often necessary to debate simulation results among distributed experts. Steering requires basic collaboration functionality.

We designed OViD to be an environment for developers and users of parallel scientific simulations. Special emphasis is laid on parallel applications that contain irregular and dynamic data structures and that run on distributed memory multiprocessor machines. We point this out since it holds a major difficulty to provide data that is irregularly distributed over parallel processes. The visualization system normally requires information about the global ordering of the data since. Post-processing tools are normally designed for sequential applications and should be independent any kind of data distribution in the running application. Global information about the data is often not available on any of the parallel processes. Also, it would be a major performance bottleneck if one process would have to collect all distributed data and send it to a visualization system or a component in between.

3.1 OViD Architecture

Fig. 2 shows the components of OViD. Central instance is the OViD server with its object manager (*omg*) and communication manager (*cmg*). The server is the layer between the parallel processes and any number of attached visualization systems. During runtime of the simulation the latter can dynamically attach to and detach from OViD.

There is one communication process (*comproc*) on each involved host (e.g. *host n*) of the parallel machine which is responsible for all communication between the parallel simulation processes and OViD. This process spawns a new thread for each communication request between one parallel process and OViD.

cmg handles all communication between the communication processes (*comproc*) on the application side in a multi-threaded manner. For each communication request of a *comproc* it creates a new thread. Therefore, communication is performed concurrently for all parallel processes and can be optimized if OViD

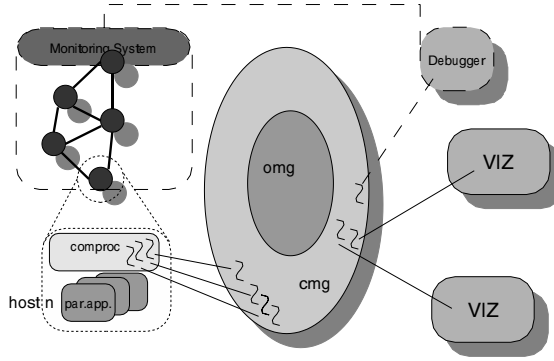


Fig. 2. OViD - Architecture

is running on a SMP machine. The multi-threading *cmg* has the advantage that the client processes are not blocked when sending their often large data sets.

Received data is stored in *omg* which can be viewed as a database. At the initialization phase all parallel processes send meta-information describing the source of the data, its type and size. This way, global information about the data objects which is needed for visualization is built in *omg*. During the running simulation each parallel process sends its part of the global data object to OViD (see Fig.3). Multiple instances (the number can be specified and altered by the user of the simulation) of the data objects can be stored in *omg*.

On the other side the different visualization tools (*VIZ*) can send requests to attach to OViD, start or re-start a simulation, get information about the available data objects, select specific data objects, send steering information, stop a simulation, detach from OViD, and more.

When steering information - which can either be a single parameter or a whole data object - is sent back to OViD, the data is distributed back to *comproc* on each host. The *comproc* creates a new thread for each communication with OViD. Steering causes synchronization among the parallel processes because it must be ensured that they receive the steering information in the same iteration. The *comproc* on each machine handles this synchronization among all *comprocs* and among the parallel processes he has to manage.

As for collaboration features, we allow multiple users at different sites to observe and steer a running simulation. If changes are performed on the data in the visualization system, we do not propagate these updates to any other visualization system. The users have to take care of consistent views themselves if theses are needed for discussions. Therefore we do not claim to be a collaboration system.

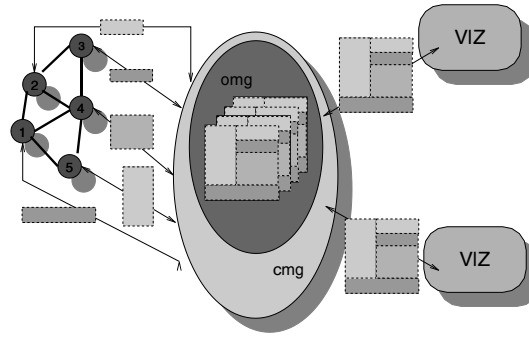


Fig. 3. Data exchange between parallel processes , OViD, and visualization system

Each *VIZ* can send requests to receive data objects to OViD. Only one *VIZ* can start the simulation, which is enforced by a token mechanism. A user can request a token at any time and is then able to send steering information to the parallel application. As long as the token is locked, no other user influences the simulation. He still can change the set of data objects he wants to observe and move back and forth among the stored data objects in OViD.

OViD provides an open interface model for parallel and distributed applications to be online interactive. There are two interfaces, one for the parallel application and one for a visualization system. An application has to be instrumented with calls to the OViD library. The data objects of interest for the user of the simulation program have to be defined as well as so-called interaction points. Interaction points are locations in the program where the defined data takes consistent values and where steering information can be imported.

It is possible to restart a parallel application with values coming from OViD instead of initial values from input files. OViD can serialize the stored data and write it to files. This features can be used for checkpointing purposes. After reading it from a file, OViD can send the data back to a parallel application. The parallel program must be modified to be able to start with data coming from OViD. Since it is possible to select the data that is transfered to OViD during a simulation run, it must be clear though that an application can only restart with values from OViD if they have been stored there previously.

The other interface to the visualization system replaces in part the tool's file interface. Information like the geometry of a simulation model will still be read from files. For online visualization the file access must be replaced by calls to OViD. All steering functionality can either be integrated into the tool, or a prototype user interface which has been developed by us can be used. The

interface model basicly allows any existing visualization tool to connect to a parallel and distributed simulation. To integrate all functionality of OViD, the source code of the visualization must be available.

One major design aspect for OViD has been platform independence. As stated in the beginning of this paper, parallel applications mostly run in heterogeneous environments. We therefore developed OViD in Java. CORBA is used as the communication platform for the whole system.

4 OViD with a Parallel CFD Simulation

Within the research project SEMPA[2], funded by the BMBF[3][5], the industrial CFD package TfC, developed and marketed by AEA Technology, GmbH was parallelized. The software can be applied to a wide range of flow problems, including laminar and turbulent viscous flows, steady or transient, isothermal or convective, incompressible or compressible (subsonic, transonic and supersonic). Application fields include pump design, turbomachines, fans, hydraulic turbines, building ventilation, pollutant transport, combustors, nuclear reactors, heat exchangers, automotive, aerodynamics, pneumatics, ballistics, projectiles, rocket motors and many more. TfC solves the mass momentum and scalar transport equations in three-dimensional space on hybrid unstructured grids. It implements a second order finite volume discretization. The system of linear equations that is assembled in the discretization phase is solved by an algebraic multigrid method (AMG).

TfC has four types of elements that are defined by their topology: hexahedral, wedge, tetrahedral and pyramid elements each of which has its specific advantages. Any combination of these element types can form a legal grid. Grid generation is the task of experienced engineers using advanced software tools. Unstructured grids cannot be represented by regular arrays. They need more complex data structures like linked lists (implemented in Fortran arrays) to store the node connectivity.

TfC is parallelized according to the SPMD model. Each parallel process executes the same program on a subset of the original data. The data (model geometry) is partitioned based on the nodes of the grid since the nodal connectivity information is implicit in the data structures of the program. Also, the solver in TfC is node based. The parallel processes only allocate memory for the local grid plus overlapping regions, never for the complete grid. Therefore, global node numbers have to be stored to map neighboring overlap nodes to the processes' local memory.

The parallelization of large numerical software packages has two main reasons. One is the execution time of the applications and the other is the problem size. Larger geometries can be computed by partitioning the problem and solving it in parallel. The performance of the parallel program must still be good when

[2] **S**oftware **E**ngineering **M**ethods for **P**arallel and Distributed **A**pplications in Scientific Computing

[3] German Federal Ministry of Education, Science, Research, and Technology

online visualization and steering is applied. The question is therefore how OViD influences the runtime of the parallel application.

We present a typical test-case of ParTfC for the evaluation of OViD. More details about the parallelization, speedup measurements, and other results of ParTfc with different test-cases can again be found in [5]. Fig. 4 shows the surface grid (4 partitions) of a spiral casing.

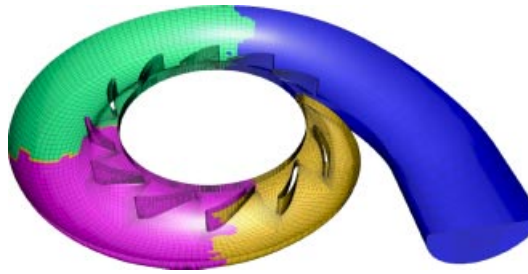


Fig. 4. Partitioned surface grid (4 partitions) for the turbulent flow in a spiral casing.

The sequential grid of an example geometry (the turbulent flow in a spiral casing) has 265.782 nodes (see also Tab. 1). We consider 4 parallel processes which are sending 4 flow variables (U, V, and W from the momentum equations as well as P from the mass conservation equation) to OViD at each time-step. If they are stored in 64bit format about 2.1 MByte of data have to be transfered by each process per time-step to OViD. We used 4 Sun Ultra10 workstations with 300MHz Ultra-SPARC IIi processors and 200MB memory connected over a 100Mbit network for our tests. OViD is running on a 4 processor Sparc Enterprise450 Ultra4 with 3GB memory.

Nr. of Processes	Nr. of Nodes	CFD-Time	Nr. of Time-steps	CFD-Time/Time-step
4	265782	1117s	5	223.4s

Table 1. Test confuguration

Tab. 1 shows in column 3 the CFD-time that is used by each parallel process for the computation of 5 time-steps. We are now measuring the time that is needed for the initialization phase of the program as well as the time to send different sizes of data to the *comproc* and OViD (see Tab. 2). Before it is sent to OViD, the data is compressed by the *comproc*s. The last column shows the time that is needed to request steering information from OViD. This information is

sent from OViD to the comproc*s* and then after a synchronization process among them sent to the parallel application processes.

Number of Bytes	Init-Phase	Send to *comproc*	Send to OViD	Receive Steering Param.
1MByte	00:01,128s	00:01,608s	00:01,954s	00:03,436s
2MByte	00:01,132s	00:2,304s	00:02,473s	00:03,529s
4MByte	00:01,147s	00:05,720s	00:05,048s	00:03,852s
8MByte	00:01,153s	00:13,08ss	00:12,127s	00:03,984s

Table 2. Measured time for different amounts of data

The time that is needed for the exchange of data either from the parallel application to OViD or vice versa must be compared with the execution time of the CFD-part. This comparison can be found in Tab. 3 where the percentage of different communication phases are compared to that time.

CFD-Time	Percentage Init	Percentage Send	Percentage Receive
223.4s	0.65%	1.03%	1.57%

Table 3. Comparison of CFD-time and OViD-time

5 Conclusion and Future Work

With OViD we have developed an interactive visualization and steering system with additional collaboration features. OViD's open interface architecture can integrate different parallel and distributed applications as well as application-specific visualization tools to an online interactive system. OViD's platform independence helps to make these applications available in heterogeneous computing environments. Its multi-threaded client/server architecture allows to observe and steer even large parallel programs with good performance. Various features like choosing between synchronization levels or changing the number and frequency of transfered data objects makes OViD a valuable environment for developers and users of high performance scientific applications. OViD can be used in a wider area of numerical simulation software than for example CUMULVS [8], SCIRun [2], or CSE [1].

Regarding parallelization, OViD can support the developer in his work. Errors in parallel programs that result from a wrong data mapping between neighboring partitions (of a grid model) can lead to a non-converging solution. Also the integration of new numerical models can lead to false numerical behavior. With OViD's online visualization such errors can be located and pursued further. Research will done on coupling OViD with a parallel debugger.

To expand the collaboration features of OViD we also started a cooperation with the group of Vaidy Sunderam at the Math and Computer Science department of Emory University. Their project CCF (Collaborative Computing Frameworks)[7] provides a suite of software systems, communications protocols, and tools that enable collaborative, computer-based cooperative work. CCF constructs a virtual work environment on multiple computer systems connected over the Internet, to form a Collaboratory. In this setting, participants interact with each other, simultaneously access and operate computer applications, refer to global data repositories or archives, collectively create and manipulate documents or other artifacts, perform computational transformations, and conduct a number of other activities via telepresence. A first step will be to provide a collaborative simulation environment for ParTfC based on CCF and OViD.

References

1. J. Mulder, J. van Wijk, and R. van Liere,: A Survey of Computational Steering Environments. Future Generation Computer Systems, Vol. 15, nr. 2, 1999.
2. C. Johnson, S. Parker, C. Hansen, G. Kindlman, and Y. Livnat: Interactive Simulation and Visualization. IEEE Computer, Vol. 32, Nr. 12, 1999.
3. T. de Fanti: Visualization in Scientific Computing. Computer Graphics, Vol. 21, 1987.
4. W. Gu et al. Falcon: Online Monitoring and Steering Parallel Programs. Concurrency: Practice & Experience, Vol. 10, No. 9, pp. 699-736, 1998
5. P. Luksch, U. Maier, S. Rathmayer, M. Weidmann, F. Unger, P. Bastian, V. Reichenberger, and A. Haas: SEMPA: Software Engineering Methods for Parallel Applications in Scientific Computing, Project Report. Research Report Series LRR-TUM (Arndt Bode, editor,)Shaker-Verlag, Vol.12, 1998.
6. P. Luksch: CFD Simulation: A Case Study in Software Engineering. High Performance Cluster Computing: Programming and Applications, Vol.2, Prentice Hall 1999
7. V. Sunderam, et al.: CCF: A Framework for Collaborative Computing. IEEE Internet Computing, Jan.2000, "http://computer.org/internet/"
8. J. A. Kohl, and P. M. Papadopoulos: Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT), Welches, OR, 1998.

A Web-Based Finite Element Meshes Partitioner and Load Balancer

Ching-Jung Liao

Department of Information Management
The Overseas Chinese Institute of Technology
407 Taichung, Taiwan, R.O.C.
cjliao@occc.edu.tw

Abstract. In this paper, we present a web-based finite element meshes partitioner and load balancer (FEMPAL). FEMPAL is an integrated tool that consists of five components, a partitioner, a load balancer, a simulator, a visualization tool, and a Web interface. Through the Web interface, other four components can be operated independently or can be cooperated with others. Besides, FEMPAL provides several demonstration examples and their corresponding mesh models that allow beginners to download and experiment. The experimental results show the practicability and usefulness of our FEMPAL.

1 Introduction

To efficiently execute a finite element application program on a distributed memory multicomputer, we need to map nodes of the corresponding mesh to processors of a distributed memory multicomputer such that each processor has the same amount of computational load and the communication among processors is minimized. Since this mapping problem is known to be *NP*-completeness , many heuristics were proposed to find satisfactory sub-optimal solutions. Based on these heuristics, many graph partitioners were developed [2], [5], [7], [9]. Among them, Jostle [9], Metis [5], and Party [7] are considered as the best graph partitioners available up-to-date. If the number of nodes of a mesh will not be increased during the execution of a finite element application program, the mapping algorithm only needs to be performed once. For an adaptive mesh application program, the number of nodes will be increased discretely due to the refinement of some finite elements during the execution of an adaptive mesh application program. This will result in load imbalance of processors. A load-balancing algorithm has to be performed many times in order to balance the computational load of processors while keeping the communication cost among processors as low as possible. To deal with the load imbalance problem of an adaptive mesh computation, many load-balancing methods have been proposed in the literature [1], [3], [4], [6], [8], [9]. Without tools support, mesh partitioning and load balancing are labor intensive and tedious. In this paper, we present a web-based finite element meshes partitioner and load balancer. FEMPAL is an integrated tool that consists of five components, a partitioner, a load balancer, a simulator, a visualization tool, and a

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 57-64, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Web interface. Besides, FEMPAL also provides several demonstration examples and their corresponding models that allow beginners to download and experiment. The design of FEMPAL is based on the criteria including easy to use, efficiency, and transparency. It is unique in using Web interface. The experimental results show that our methods produced 3% to 13% fewer cut-edges and reduced simulation time by 0.1% to 0.3%. The rest of the paper is organized as follows. The related work will be given in Section 2. In Section 3, the FEMPAL will be described in details. In Section 4, some experimental results of using FEMPAL will be presented.

2 Related Work

Many methods have been proposed to deal with the partitioning/mapping problems of irregular graphs on distributed memory multicomputers in the literature. These methods were implemented in several graph partition libraries, such as Jostle, Metis, and Party, etc., to solve graph partition problems. For the load imbalance problem of adaptive mesh computations, many load-balancing algorithms can be used to balance the load of processors. Hu and Blake [4] proposed a *direct diffusion* method that computes the diffusion solution by using an unsteady heat conduction equation while optimally minimizing the Euclidean norm of the data movement. They proved that a diffusion solution can be found by solving the linear equation. Horton [3] proposed a *multilevel diffusion* method by recursively bisecting a graph into two subgraphs and balancing the load of the two subgraphs. This method assumes that the graph can be recursively bisected into two connected graphs. Schloegel *et al*. [8] also proposed a multilevel diffusion scheme to construct a new partition of the graph incrementally. C. Walshaw *et al*. [9] implemented a parallel partitioner and a direct diffusion repartitioner in Jostle that is based on the diffusion solver proposed by Hu and Blake [4]. Although several graph partitioning and load-balancing methods have been implemented as tools or libraries [5], [7], [9], none of them has offered its Web interface. Our FEMPAL is unique in using Web interface and high level support to users.

3 The System Structure of FEMPAL

The system structure of FEMPAL consists of five components, a partitioner, a load balancer, a simulator, a visualization tool, and a Web interface. Users can upload the finite element mesh data and get the running results on any Web browsers. Through the Web interface, other four components can be operated independently or can be cooperated with others. In the following, we will describe them in details.

3.1 The Partitioner

In the partitioner, we provide three partitioning methods, Jostle/DDM, Metis/DDM, and Party/DDM. These methods were implemented based on the best algorithms provided in Jostle, Metis and Party, respectively, with the dynamic diffusion optimization method (DDM) [2]. In FEMPAL, we provide five 2D and two 3D finite ele-

ment demo meshes. The outputs of the partitioner are a partitioned mesh file and partitioned results. In a partitioned mesh file, a number j in line i indicates that node i belongs to processor j. The partitioned results include the load balancing degree and the total cut-edges of a partitioned mesh.

3.2 The Load Balancer

In the load balancer, we provide two load-balancing methods, the prefix code matching parallel load-balancing (PCMPLB) method [1] and the binomial tree-based parallel load-balancing (BINOTPLB) method [6]. In the load balancer, users can also use the partitioned finite element demo mesh model provided by FEMPAL. In this case, the inputs are the load imbalance degree and the number of processors. The outputs of the load balancer are a load-balanced mesh file and the load balancing results. The load balancing results include the load balancing degree and the total cut-edges.

3.3 The Simulator

The simulator provides a simulated distributed memory multicomputer for the performance evaluation of a partitioned mesh. The execution time of a mesh on a P-processor distributed memory multicomputer under a particular mapping/load-balancing method L_i can be defined as follows:

$$T_{par}(L_i) = max\{T_{comp}(L_i, P_j) + T_{comm}(L_i, P_j)\} \ , \quad \textbf{(1)}$$

where $T_{par}(L_i)$ is the execution time of an mesh on a distributed memory multicomputer under L_i, $T_{comp}(L_i, P_j)$ is the computation cost of processor P_j under L_i, and $T_{comm}(L_i, P_j)$ is the communication cost of processor P_j under L_i, where $j = 0, ..., P-1$.

The cost model used in Equation 1 is assuming a synchronous communication mode in which each processor goes through a computation phase followed by a communication phase. Therefore, the computation cost of processor P_j under a mapping/load-balancing method L_i can be defined as follows:

$$T_{comp}(L_i, P_j) = S \times load_i(P_j) \times T_{task}, \quad \textbf{(2)}$$

where S is the number of iterations performed by a finite element method, $load_i(P_j)$ is the number of nodes of an finite element mesh assigned to processor P_j, and T_{task} is the time for a processor to execute tasks of a node.

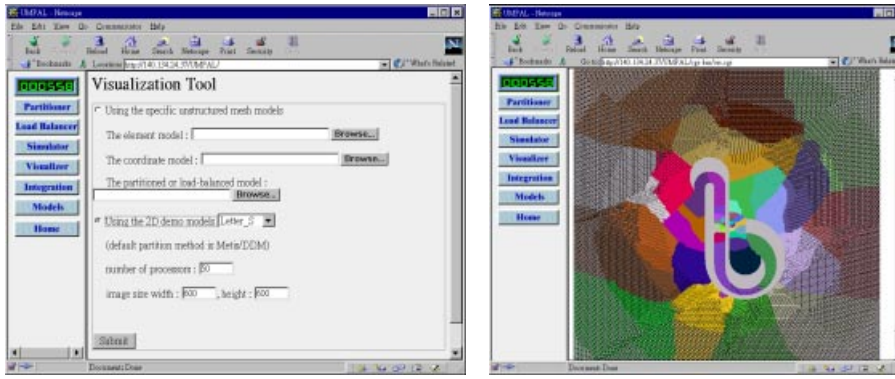
For the communication model, we assume a synchronous communication mode and every two processors can communicate with each other in one step. In general, it is possible to overlap communication with computation. In this case, $T_{comm}(L_i, P_j)$ may not always reflect the true communication cost since it would be partially overlapped with computation. However, $T_{comm}(L_i, P_j)$ should provide a good estimate for the communication cost. Since we use a synchronous communication mode, $T_{comm}(L_i, P_j)$ can be defined as follows:

$$T_{comm}(L_i, P_j) = S \times (\delta \times T_{setup} + \phi \times T_c), \quad \textbf{(3)}$$

where S is the number of iterations performed by a finite element method, δ is the number of processors that processor P_j has to send data to in each iteration, T_{setup} is the setup time of the I/O channel, ϕ is the total number of data that processor P_j has to send out in each iteration, and T_c is the data transmission time of the I/O channel per byte. To use the simulator, users need to input the partitioned or load-balanced mesh file and the values of S, T_{setup}, T_c, T_{task}, and the number of bytes sent by a finite element node to its neighbor nodes. The outputs of the simulator are the execution time of the mesh on a simulated distributed memory multicomputer and the total cut-edges of a partitioned mesh.

3.4 The Visualization Tool

FEMPAL also provides a visualization tool for users to visual the partitioned finite element mesh. The inputs of the visualization tool are files of the coordinate model, the element model, and the partitioned finite element mesh models of an finite element mesh, and the size of an image. For the coordinate model file of an finite element mesh, line 1 specifies the number of nodes in an finite element mesh. Line 2 specifies the coordinate of node 1. Line 3 specifies the coordinate of node 2, and so on. Fig. 1(a) shows the Web page of the visualization tool. After rendering, a Web browser displays the finite element mesh with different colors, and each color represents one processor. Fig. 1(b) shows the rendering result of the test sample *Letter_S*.



(a) (b)

Fig. 1. (a) The Web page of the visualization tool. (b) The rendering result of *Letter_S*.

3.5 The Web Interface

The Web interface provides a mean for users to use FEMPAL through Internet and integrates other four parts. The Web interface consists of two parts, an HTML interface and a CGI interface. The HTML interface provides Web pages for users to input requests from Web browsers. The CGI interface is responsible for handling requests

of users. Through the Web interface, other four components can be operated independently or can be cooperated with others. As users operate each component independently, the Web interface passes the requests to the corresponding component. The corresponding component will then process the requests and produce output results.

3.6 The Implementation of FEMPAL

In order to support standard WWW browsers, the front end is coded in HTML with CGI. The CGI interface is implemented in Perl language. The CGI interface receives the data and parameters from the forms of the HTML interface. It then calls external tools to handle requests. The tools of FEMPAL, partitioner, balancer, and simulator are coded in C language. They receive the parameters from the CGI interface and use the specified methods (functions) to process requests of users.

To support an interactive visualization tool, the client/server software architecture is used in FEMPAL. In the client side, a Java Applet is implemented to display images rendered by server. In the server side, a Java server-let is implemented as a Java Application. The Java server-let renders image with specific image size and finite element mesh models. As the server finishes its rendering work, it sends the final image to client side and users can see the final image from users' Web browsers.

4 Experience and Experimental Results

In this section, we will present some experimental results for finite element meshes by using the partitioner, the load balancer, and the simulator of FEMPAL through a Web browser.

4.1 Experimental Results for the Partitioner

To evaluate the performance of Jostle/DDM, ML*k*P/DDM, and Party/DDM, three 2D and two 3D finite element meshes are used as test samples. The number of nodes, the number of elements, and the number of edges of these five finite element meshes are given in Table 1.

Table 2 shows the total cut-edges of the three methods with their counterparts for the test meshes on 70 processors. The total cut-edges of Jostle, Metis, and Party were obtained by running these three partitioners with default values. The load imbalance degree allowed by Jostle, Metis, and Party are 3%, 5%, and 5%, respectively. The total cut-edges of the three methods were obtained by applying the dynamic diffusion optimization method (DDM) to the partitioned results of Jostle, Metis, and Party, respectively. The three methods guarantee that the load among partitioned modules is fully balanced. From Table 2, we can see that the total cut-edges produced by the methods provided in the partitioner are less than those of their counterparts. The DDM produced 1% to 6% fewer total cut-edges in most of the test cases.

Table 1. The number of nodes, elements, and edges of the test samples.

Samples	#node	#element	#edges
Hook	80494	158979	239471
Letter_S	106215	126569	316221
Truss	57081	91968	169518
Femur	477378	953344	1430784
Tibia	557058	1114112	1671168

Table 2. The total cut-edges of the methods provided in the partitioner and their counterparts for three 2D and two 3D finite element meshes on 70 processors.

Model	Method					
	Jostle	Jostle/DDM	Metis	Metis/DDM	Party	Party/DDM
Hook	7588	7508 (-1%)	7680	7621 (-1%)	8315	8202(-1%)
Letter_S	9109	8732 (-4%)	8949	8791 (2%)	9771	9441(-3%)
Truss	7100	6757 (-5%)	7153	6854 (-4%)	7520	7302(-3%)
Femur	23982	22896 (-5%)	23785	23282 (-2%)	23004	22967(-0.2%)
Tibia	26662	24323 (-10%)	26356	24887 (-6%)	25442	25230(-1%)

4.2 Experimental Results for the Load Balancer

To evaluate the performance of PCMPLB and BINOTPLB methods provided in the load balancer, we compare these two methods with the direct diffusion (DD) method and the multilevel diffusion (MD) method. We modified the multilevel *k*-way partitioning (ML*k*P) program provided in Metis to generate the desired test samples. The methods provided in the load balancer guarantee that the load among partitioned modules will be fully balanced while the DD and MD methods do not. Table 3 shows the total cut-edges produced by DD, MD, PCMPLB, and BINOTPLB for the test sample *Tibia* on 50 processors. From Table 3, we can see that the methods provided in the load balancer outperform the DD and MD methods. The PCMPLB and BINOTPLB methods produced 9% to 13% fewer total cut-edges than the DD and MD methods. The load balancing results of PCMPLB and BINOTPLB depend on the test samples. It is difficult to tell that which one performs better than the other for a given partitioned finite element mesh. However, one can select both methods in the load balancer, see the load balancing results, and choose the best one.

Table 3. The total cut-edges produced by DD, MD, PCMPLB, and BINOTPLB for the test sample *Tibia* on 50 processors.

Load imbalance	Method							
	DD	MD	PCMPLB			BINOTPLB		
3%	22530	22395	19884	(-13%)	(-13%)	19868	(-13%)	(-13%)
5%	22388	22320	20398	(-10%)	(-9%)	20060	(-12%)	(-11%)
10%	22268	22138	20411	(-9%)	(-9%)	20381	(-9%)	(-9%)

4.3 Experience with the Simulator

In this experimental test, we use the simulator to simulate the execution of a parallel Laplace solver on a 70-processor SP2 parallel machine. According to [1], the values of T_{setup}, T_c, and T_{task} are 46μs, 0.035μs, and 350μs, respectively. Each finite element node needs to send 40 bytes to its neighbor nodes. The number of iterations performed by a Laplace solver is set to 10000. Table 4 shows the simulation results of test samples under different partitioning methods provided in the partitioner on a simulated 70-processor SP2 parallel machine. For the performance comparison, we also include the simulation results of test samples under Jostle, Metis, and Party in Table 4. From Table 2 and Table 4, we can see that, in general, the smaller the total cut-edges, the less the execution time. The simulation result may provide a reference for a user to choose a right method for a given mesh.

Table 4. The simulation results of test samples under different partitioning methods provided in the partitioner on a simulated 70-processor SP2 parallel machine. (Time : second)

	Method					
Model	Jostle	Jostle/DDM	Metis	Metis/DDM	Party	Party/DDM
Truss	2870.578	2861.318	2861.836	2861.374	2864.178	2863.272
Letter_S	5328.878	5319.074	5318.698	5318.698	5328.004	5326.274
Hook	4042.268	4030.390	4030.614	4030.628	4035.036	4033.320
Tibia	27868.974	27862.126	27862.576	27862.100	27865.982	27865.940
Femur	23905.270	23878.790	23878.962	23878.990	23879.788	23883.260

5 Conclusions and Future Work

In this paper, we have presented a software tool, FEMPAL, to process the partitioning and load balancing problems for the finite element meshes on World Wide Web. Users can use FEMPAL by accessing its Internet location, http://www.occc.edu.tw/~cjliao. FEMPAL is an integrated tool that consists of five components, a partitioner, a load balancer, a simulator, a visualization tool, and a Web interface. The design of FEMPAL is based on the criteria including easy to use, efficiency, and transparency. The experimental results show the practicability and usefulness of FEMPAL. The integration of different methods into FEMPAL has made the experiments and simulations of parallel programs very simple and cost effective. FEMPAL offers a very high level and user friendly interface. Besides, the demonstration examples can educate beginners on how to apply finite element method to solve parallel problems. There is one typical shortage for tools on WWW, which is the downgrade of performance when multiple requests have been requested. To solve this problem, we can either execute FEMPAL on a more powerful computer or execute FEMPAL on a cluster of machines. In the future, the next version of FEMPAL will execute on a PC clusters environment.

6 Acknowledgments

The author would like to thank Dr. Robert Preis, Professor Karypis, and Professor Chris Walshaw for providing the Party, the Metis, and Jostle software packages.

I want to thank Professor Yeh-Ching Chung for his advise in paper writing, and Jen-Chih Yu for writing the web-based interface program.

I also would like to thank Prof. Dr. Arndt Bode and Prof. Dr. Thomas Ludwig for their supervision and take care when I stayed in TUM Germany.

References

1. Chung, Y.C., Liao, C.J., Yang, D.L.: A Prefix Code Matching Parallel Load-Balancing Method for Solution-Adaptive Unstructured Finite Element Graphs on Distributed Memory Multicomputers. The Journal of Supercomputing. Vol. 15. 1 (2000) 25-49
2. Chung, Y.C., Liao, C.J., Chen, C.C., Yang, D.L.: A Dynamic Diffusion Optimization Method for Irregular Finite Element Graph Partitioning. to appear in The Journal of Supercomputing.
3. Horton, G.: A Multi-level Diffusion Method for Dynamic Load Balancing. Parallel Computing. Vol. 19. (1993) 209-218
4. Hu, Y.F., Blake, R.J.: An Optimal Dynamic Load Balancing Algorithm. Technical Report DL-P-95-011, Daresbury Laboratory, Warrington, UK (1995)
5. Karypis, G., Kumar, V.: Multilevel *k*-way Partitioning Scheme for Irregular Graphs. Journal of Parallel and Distributed Computing, Vol. 48. No. 1 (1998) 96-129
6. Liao, C.J., Chung, Y.C.: A Binomial Tree-Based Parallel Load-Balancing Methods for Solution-Adaptive Unstructured Finite Element Graphs on Distributed Memory Multicomputers. Proceedings of 1998 International Conference on Parallel CFD. (1998)
7. Preis, R., Diekmann, R.: The PARTY Partitioning - Library, User Guide - Version 1.1. Technical Report tr-rsfb-96-024, University of Paderborn, Germany (1996)
8. Schloegel, K., Karypis, G., Kumar, V.: Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. Journal of Parallel and Distributed Computing. Vol. 47. 2 (1997) 109-124
9. Walshaw, C.H., Cross, M., Everett, M.G.: Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. Journal of Parallel and Distributed Computing. Vol. 47. 2 (1997) 102-108

A Framework for an Interoperable Tool Environment

Radu Prodan[1] and John M. Kewley[2]

[1] Institut für Informatik, Universität Basel,
Mittlere Strasse 142, CH-4056 Basel, Switzerland
prodan@ifi.unibas.ch
[2] Centro Svizzero di Calcolo Scientifico (CSCS),
Via Cantonale, CH-6928 Manno, Switzerland.
kewley@cscs.ch

Abstract. Software engineering tools are indispensable for parallel and distributed program development, yet the desire to combine them to provide enhanced functionality has still to be realised. Existing tool environments integrate a number of tools, but do not achieve interoperability and lack extensibility. Integration of new tools can necessitate the redesign of the whole system. We describe the FIRST tool framework, its initial tool-set, classify different types of tool interaction and describe a variety of tool scenarios which are being used to investigate tool interoperability.

1 Introduction

A variety of software engineering tools are now available for removing bugs and identifying performance problems, however these tools rely on different compilation options and have no way of interoperating. Integrated tool environments containing several tools do offer some degree of interoperability; they do, however, have the disadvantage that the set of tools provided is fixed and the tools are typically inter-dependent, interacting through internal proprietary interfaces. The result is a more complex tool which combines the functionality of the integrated tools, but lacks true interoperability and is closed to further extension.

The FIRST[1] project, a collaboration between the University of Basel and the Swiss Center for Scientific Computing, defines an extensible framework [1] for the development of interoperable software tools. Tool interaction is enhanced by its high-level interoperable tool services [2], which provide a tool development API. Figure 1 shows the object-oriented architecture for FIRST (see [2] for further information). The Process Monitoring Layer (PML) handles the platform dependent aspects of the tool; it attaches to an application and extracts run-time information using the Dyninst [3] dynamic instrumentation library. The Tool Services Layer utilises this functionality to present a set of high-level tool services

[1] Framework for Interoperable Resources, Services, and Tools.
Supported by grant 21–49550.96 from the Swiss National Science Foundation.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 65–69, 2000.
© Springer-Verlag Berlin Heidelberg 2000

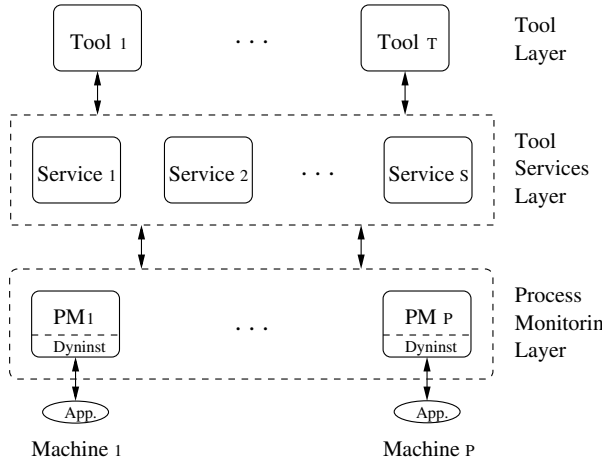


Fig. 1. The FIRST Architecture.

to the tool developer: *Information Requests* provide information about the state of the running application. *Performance Metrics* specify run-time data (e.g., counters and timers), to be collected by the PML; *Breakpoints* set normal or conditional breakpoints in the application; and *Notifications* inform the requester when certain events have occurred.

This paper classifies different types of tool interaction and describes a variety of tool scenarios (Sec. 3) to be used to investigate tool interoperability.

2 Initial Toolset

The FIRST project focuses on the design and development of an open set of tool services, however to prove the viability of the framework, a set of interoperable tools has been implemented. These tools operate on unmodified binaries at run-time and can be used to monitor both user and system code even when there is no source code available. There is no dependence on compilation options or flags. The intention is to develop simple tools that, as well as demonstrate the concepts of the FIRST approach, can then be used as building blocks that interoperate to reproduce the functionality of traditionally more complex tools.

`Object Code Browser (OCB)` is a browsing tool, which graphically displays the object code structure of a given process. It can also be used alongside other tools for selecting the functions to be instrumented (Sec. 3.1).

`1st_top` inspired by the Unix administration tool `top`, was designed to display the top n functions in terms of the number of times they were called, or in terms of execution time.

`1st_trace` in the style of the Unix software tool `truss`, traces the functions executed by the application as it executes.

`1st_prof` like the Unix tool `prof` displays the call-graph profile data, by timing and counting function calls.

`1st_cov` imitates the Unix tool `tcov` to produce a test coverage analysis on a function basis.

`1st_debug` is a traditional debugger, like `dbx`, with the capabilities of controlling processes, acquiring the object-structure of the application, viewing variables, and adding breakpoints.

This set of tools will be extended to cover more specific parallel programming aspects (e.g., deadlock detector, message queue visualiser, distributed data visualiser, and memory access tool).

3 Tool Interoperability Scenarios

One of the main issues addressed by the framework is to provide an open environment for interoperability and to explore how various tools can co-operate in order to gain synergy. FIRST considers several types of tool interaction [4]:

Direct Interaction assumes direct communication between tools. They are defined by the tool's design and implementation, and happen exclusively within the tool layer; they are not performed by the framework.

Indirect Interaction is a more advanced interaction, mediated by the framework via its services. It requires no work or knowledge from the tools (which might not even know about each others existence). It occurs when the FIRST tool services interact with each other on behalf of the tools. These indirect interactions can occur in several ways:

Co-existence when multiple tools operate simultaneously, each on its own parallel application, sharing FIRST services,

Process Sharing when multiple tools attach and instrument the same application process at the same time. This kind of interoperability is consistently handled by the framework through its co-ordinated services.

Instrumentation Sharing when tools share instrumentation snippets while monitoring the same application process to minimise the probe-effect. This is automatically handled by FIRST.

Resource Locking when tools require exclusive access to a specific resource. For example a tool could ask for a lock on a certain application resource (process or function) so that it may perform some accurate timing. No other tool would be allowed to instrument that resource, but could instead use the existing timers (instrumentation sharing).

3.1 Interaction with a Browser

Many tools need to display the resource hierarchy [5] of an application. This includes object code structure (modules, functions and instrumentation points),

machines, processes and messages. Since it is inefficient for every tool to independently provide such functionality, the responsibility can be given to a single tool, the OCB (Sec. 2). This can display the resource hierarchy of an application and also be used to specify which resources are to be used when another FIRST tool is run. For instance by selecting a set of functions and then running `1st_prof` with no other arguments, the selected functions will be monitored.

3.2 Computational Steering

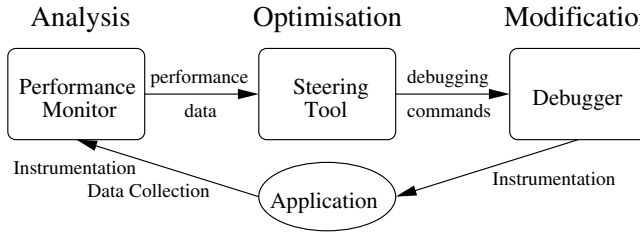


Fig. 2. The Steering Configuration.

The FIRST computational steering tool will directly interact with two other tools: a performance monitor which collects performance data and presents it to the steering tool and a debugger which dynamically carries out the optimisations chosen by the steering tool. The use of dynamic instrumentation enables the steering process to take place dynamically, within one application run, hence there is no need to rerun the application every time a modification is made.

The interoperability between the three tools is mixed. The steering tool interacts directly with the other two. The performance monitor and the debugger interact indirectly, by concurrently manipulating the same application process (using the same monitoring service).

3.3 Interaction with a Debugger

The interaction of tools with a run-time interactive debugger requires special attention due to its special nature of manipulating and changing the process's execution. Two distinct indirect interactions (process sharing) are envisaged:

Consistent Display Providing a consistent display is an important task no run-time tool can avoid. When multiple tools are concurrently monitoring the same processes, this issue becomes problematic since each tool's display depends not only on its own activity, but also on that of the others'.

When a visualiser interoperates with a debugger the following interactions could happen:

- if the debugger stops the program's execution, the visualiser needs to update its display to show this;
- if the debugger changed the value of a variable, for consistency, the visualiser should update its display with the new value;
- if the debugger loaded a shared library in the application, or replaced a call to a function, the OCB must change its code hierarchy accordingly.

Timing Another important interaction can happen between a performance tool and a debugger. For example, a debugger could choose to stop a process whilst the performance tool is performing some timing operations on it. Whilst the user and system time stop together with the process, the wall-time keeps running. The framework takes care of this and stops the wall-timers whenever the application is artificially suspended by the debugger.

4 Conclusion and Future Work

This paper has introduced an open extensible framework for developing an interoperable tool environment. The requirements for a set of interoperable tools were described along with a brief introduction to the software tools currently available within FIRST. The notion of tool interoperability was analysed and the resulting classification of tool interactions was presented. Interoperability scenarios were presented to investigate the various ways tools can interact to gain synergy.

These scenarios for interoperating tools will be the subject of future work to validate the capacity of the framework for interoperability.

References

[1] Radu Prodan and John M. Kewley. FIRST: A Framework for Interoperable Resources, Services, and Tools. In H. R. Arabnia, editor, *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (Las Vegas, Nevada, USA)*, volume 4. CSREA Press, June 1999.

[2] John M. Kewley and Radu Prodan. A Distributed Object-Oriented Framework for Tool Development. In *34th International Conf. on Technology of Object-Oriented Languages and Systems (TOOLS USA)*. IEEE Computer Society Press, 2000.

[3] Jeffrey K. Hollingsworth and Bryan Buck. DyninstAPI Programmer's Guide. Manual, University of Maryland, College Park, MD 20742, September 1998.

[4] Roland Wismüller. Interoperability Support in the Distributed Monitoring System OCM. In R. Wyrzykowski et al., editor, *Proceedings of the 3rd International Conf. on Parallel Processing and Applied Mathematics (PPAM'99)*, volume 4, pages 77–91. Technical University of Czestochowa, Poland, September 1999.

[5] Barton P. Miller, R. Bruce Irvin, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.

ToolBlocks: An Infrastructure for the Construction of Memory Hierarchy Analysis Tools

Timothy Sherwood and Brad Calder

University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

Abstract. In an era dominated by the rapid development of faster and cheaper processors it is difficult both for the application developer and system architect to make intelligent decisions about application interaction with system memory hierarchy. We present ToolBlocks, an object oriented system for the rapid development of memory hierarchy models and analysis. With this system, a user may build cache and prefetching models and insert complex analysis and visualization tools to monitor their performance.

1 Introduction

The last ten years have seen incredible advances in all aspects of computer design and use. As the hardware changes rapidly, so must the tools used to optimize applications for it. It is not uncommon to change cache sizes and even move caches from off chip to on chip within the same chip set. Nor is it uncommon for comparable processors or DSPs to have wildly different prefetching and local memory structures. Given an application it can be a daunting task to analyze the existing DSPs and to choose one that will best fit your price/performance goals. In addition, many researchers believe that future processors will be highly configurable by the users, further blurring the line between application tuning and hardware issues. All of these issues make it increasingly difficult to build an suite of tools to handle the problem.

To address this problem we have developed *ToolBlocks*, an object oriented system for the rapid development of memory hierarchy models and tools. With this system a user may easily and quickly modify simulated memory hierarchy layout, link in preexisting or custom analysis and visualization code, and analyze real programs all within a span of hours rather than weeks. From our experience we found that there are three design rules necessary for building any successful system for memory hierarchy analysis.

1. Extendibility: Both the models and analysis must be easily extendible to support the ever changing platforms and new analysis techniques.
2. Efficiency: Most operations should be able to be done with a minimal amount of coding (if any) in a small amount of time.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 70–74, 2000.
© Springer-Verlag Berlin Heidelberg 2000

3. Visualization: Visualization is key to understanding complex systems and the memory hierarchy is no different.

Noticeably missing from the list is performance. This has been the primary objective of most other memory hierarchy simulators [1,2,3], and while we find that reasonable performance is necessary, it should not be sought at the sacrifice of any of the other design rules (most notably extendibility).

ToolBlocks allows for extendibility through it's object oriented interface. New models and analysis can be quickly prototyped, inserted into the hierarchy, tested and used. Efficiency is achieved through a set of already implemented models, analysis and control blocks that can be configured rapidly by a user to gather a wide range of information. To support visualization the system hooks directly into a custom X-windows visualization program which can be used to analyze data post-mortem or dynamically over the execution of an interactive program.

2 System Overview

ToolBlocks can be driven from either a trace, binary modification tool, or simulator. It is intended to be an add on to, not a replacement for, lower level tools such as ATOM [4] and DynInst [5]. It was written to make memory hierarchy research and cross platform application tuning more fruitful and to reduce redundant effort. Figure 1 shows how tool blocks fits into the overall scheme of analysis.

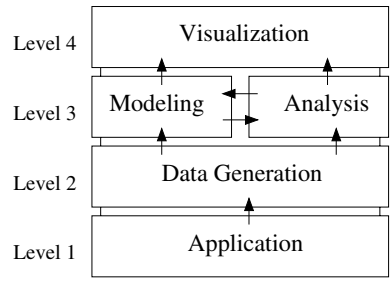


Fig. 1. Typical flow of data in an analysis tool. Data is gathered from the application which is used to do simulations and analysis whose results are then visualized

At the bottom level we see the application itself. It is here that all analysis must start. Level 2 is where data is gathered either by a tracing tool, binary modification tool or simulation. Level 3 is where the system is modeled, statistics are gathered, and analysis is done. At the top level data is visualized by the end user. ToolBlocks does the modeling and analysis of level 3, and provides some visualization.

The ToolBlocks system is completely object oriented. The classes, or *blocks*, link together through *reference streams*. From this, a set of blocks called a *block stack* is formed. The block stack is the final tool and consists of both the memory hierarchy simulator and the analysis. At the top of the block stack is a terminator, and at the bottom (the leafs) are the inputs. The stack takes input at the bottom and sends it through the chain of blocks until it reaches a terminator. Figure 2 is a simple example of a block stack.

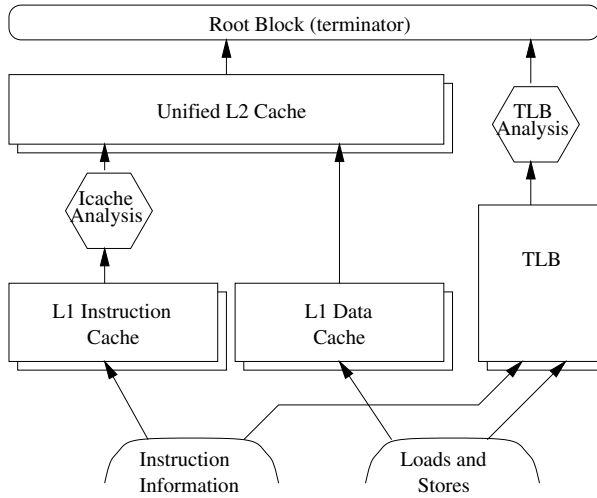


Fig. 2. The basic memory hierarchy block stack used in this paper. Note the terminating `RootBlock`. The inputs at the bottom may be generated by trace or binary modification. The hexagons are analysis, such as source code tracking or conflict detection.

The class hierarchy is intentionally quite simple to support ease of extendibility. There is a base class, called `BaseBlock`, from which all blocks inherit. The base block contains no information, it simply defines the most rudimentary interface. From this there are three major types of blocks defined: model blocks, control blocks, and analysis blocks.

Model blocks represent the hardware structures of the simulated architecture, such as cache structures and prefetching engines. These blocks, when assembled correctly, form the base hardware model to be simulated.

Control blocks modify streams in simple ways to aid the construction of useful block stacks. The simplest of the control blocks is the root block, which terminates the stack by handling all inputs but creating no outputs. However there are other blocks which can be used to provide user configurability without having to code anything. For example filter, split and switch blocks.

The analysis blocks are the most interesting part of the ToolBlocks system. The analysis blocks are inserted into streams but have no effect on them, they simply pass data along, up to the next level without any modifications. The analysis blocks are used to look at the characteristics of each stream so that a better understanding of the traffic at that level can be gained. There are currently four analysis routines, `TraceBlock` for generating traces, `PerPcBlock` for tracking memory behavior back to source code, `HistogramBlock` for dividing up the data into buckets, and `ViewBlock` for generating a visual representation of the data. These analysis routines could further be linked into other available visualization tools.

The total slowdown of program execution varies depending on the block stack, but is typically between 15x and 50x for a reasonable cache hierarchy and a modest amount of analysis and all the ATOM code inserted into the original binary.

2.1 Example Output

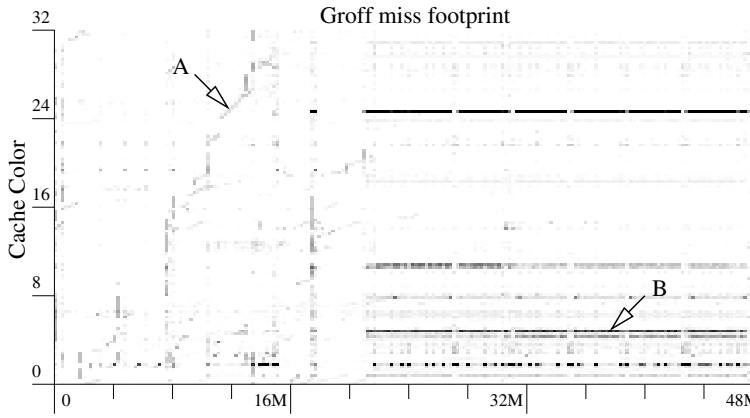


Fig. 3. Original footprint for the application Groff. The x axis is in instructions executed (millions) and the y axis is the division by cache color. Note the streaming behavior seen at point A, and the conflict behavior at point B.

Having now seen an overview of how the system is constructed, we now present an example tool and show how it was used to conduct memory hierarchy research. The tool we present is a simple use of the cache model with a `ViewBlock` added to allow analysis of L2 cache misses. The memory hierarchy is a split virtually indexed L1, and a virtually indexed unified L2. On top of the L2 is a visualization block allowing all cache misses going to main memory to be seen.

Figure 3 shows the memory footprint of the C++ program `groff` taken for a 256K L2 cache. On the X axis is the number of instructions executed, and

on the Y axis is a slice of the data cache. Each horizontal row in the picture is a cache line. The darker it is the more cache misses per instruction. As can be seen, there are two major types of misses prevalent in this application, streaming misses (at point A) and conflict misses (at point B).

The streaming capacity/compulsory misses, as pointed to by arrow A, are easily seen as angled or vertical lines because as memory is walked through, sequential cache lines are touched in order. Conflict misses on the other hand are characterized as long horizontal lines. As two or more sections of memory fight for the same cache sets, they keep kicking each other out, which results in cache sets that almost always miss.

From this data, and through the use of the PerPC block, these misses can be tracked back to the source code that causes them in a matter of minutes. The user could then change the source code or the page mapping to avoid this.

3 Conclusion

In this paper we present ToolBlocks as an infrastructure for building memory hierarchy analysis tools for application tuning, architecture research, and reconfigurable computing. Memory hierarchy tools must provide ease of extension to support a rapidly changing development environment and we describe how an powerful and extendible memory hierarchy tool can be built from the primitives of models, analysis, and control blocks.

We find that by tightly coupling the analysis and modeling, and by the promotion of analysis blocks to first class membership, a very simple interface can provide a large set of useful functions. The ToolBlocks system is a direct result of work in both conventional and reconfigurable memory hierarchy research and is currently being used tested by the Compiler and Architecture and MORPH/AMRM groups at UC San Diego. You can retrieve a version of ToolBlocks from `http://www-cse.ucsd.edu/ groups/ pacl/ tools/ toolblocks.html`. This research was supported by DARPA/ITO under contract number DABT63-98-C-0045.

References

1. Sugumar, R., Abraham, S.: Cheeta Cache Simulator, From University of Michigan.
2. Hill, M., Smith, A.: Evaluating Associativity in CPU Caches. IEEE Trans. on Computers, C-38, **12**, December 1989, p.1612–1630.
3. Gee, J., Hill, M., Pnevmatikatos, D., Smith, A. Cache Performance of the SPEC Benchmark Suite. IEEE Micro, August 1993, **3**, 2.
4. Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the Conference on Programming Language Design and Implementation, pages 196-205. ACM, 1994.
5. Hollingsworth, J., Miller, B., Cargille, J.: Dynamic Program Instrumentation for Scalable Performance Tools In the Proceedings of 1994 Scalable High Performance Computing Conference, May 1994.

A Preliminary Evaluation of FINESSE, a Feedback-Guided Performance Enhancement System

Nandini Mukherjee, Graham D. Riley, and John R. Gurd

Centre for Novel Computing,
Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester, UK
{nandini, griley, john}@cs.man.ac.uk
http://www.cs.man.ac.uk/cnc

Abstract. Automatic parallelisation tools implement sophisticated tactics for analysing and transforming application codes but, typically, their "one-step" strategy for finding a "good" parallel implementation remains naïve. In contrast, successful experts at parallelisation explore interesting parts of the space of possible implementations. FINESSE is an environment which supports this exploration by automating many of the tedious steps associated with experiment planning and execution and with the analysis of performance. FINESSE also harnesses the sophisticated techniques provided by automatic compilation tools to provide feedback to the user which can be used to guide the exploration. This paper briefly describes FINESSE and presents evidence for its effectiveness. The latter has been gathered from the experiences of a small number of users, both expert and non-expert, during their parallelisation efforts on a set of compute-intensive, kernel test codes. The evidence lends support to the hypothesis that users of FINESSE can find implementations that achieve performance comparable with that achieved by expert programmers, but in a shorter time.

1 Introduction

FINESSE is an environment which provides semi-automatic support for a user attempting to improve the performance of a parallel program executing on a single-address-space computer, such as the SGI Origin 2000 or the SGI Challenge. FINESSE aims to improve the productivity of programmers by automating many of the tedious tasks involved in exploring the space of possible parallel implementations of a program [18]. Such tasks include: managing the versions of the program (*implementations*) created during the exploration; designing and executing the instrumentation experiments required to gather run-time data; and informative analysis of the resulting performance. Moreover, the environment

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 75–85, 2000.
© Springer-Verlag Berlin Heidelberg 2000

provides feedback designed to focus the user's attention on the performance-critical components of the application. This feedback currently includes automatic interpretation of performance analysis data and it is planned to include recommendations of program transformations to be applied, if desired, by automatic restructuring tools integrated into the environment. This enables a user efficiently to take advantage of the sophisticated tactics employed by restructurers, and to apply them as part of a parallelisation strategy that is more sophisticated than the naïve "one-step" strategy typically used by such tools [10].

FINESSE has been described in detail elsewhere [9]. This paper presents an initial evaluation of the environment based on the experiences of a small number of users, both expert and non-expert, while parallelising a set of six kernel test codes on the SGI Origin 2000 and SGI Challenge platforms. In each case, parallelisation was conducted in three ways: (1) manually (following the manual parallelisation scheme, *Man*, presented in [18]); (2) with the aid of some well-known automatic compilation systems (*PFA* [16], *Polaris* (Pol) [17] and *Parafrase* (Pfr) [14]); and (3) with the support of FINESSE. Experience with one test code, known as *SPEC* (and referred to later as **SP**) — a kernel for Legendre transforms used in numerical weather prediction [19] — is considered in some depth, and summary results for all six test codes are presented.

The paper is organised as follows: Section 2 provides a brief overview of FINESSE. Section 3 describes the application kernels and the experimental method used in the evaluations. Sections 4 and 5 report the experiences gained with the **SP** code using each of the parallelisation processes described above. Section 6 presents results for other test codes. Section 7 discusses related work and Section 8 concludes.

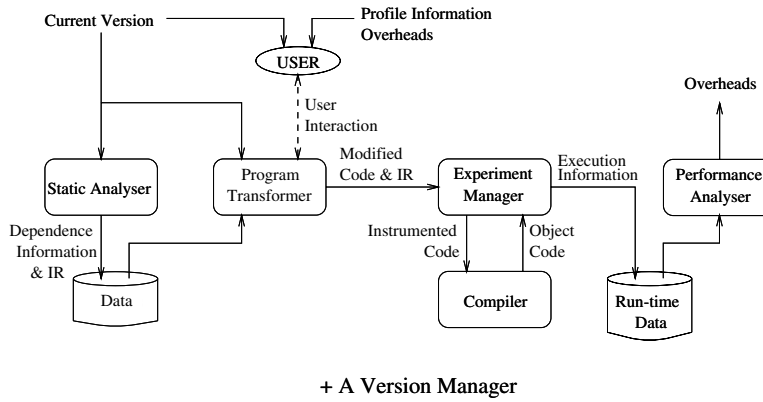


Fig. 1. Overview of FINESSE.

2 Overview of FINESSE

Performance improvement in FINESSE involves repeated analysis and transformation of a series of implementations of the program, guided by feedback information based on both the program structure and run-time execution data. The FINESSE environment is depicted in Figure 1; it comprises the following:

- A **version manager**, which keeps track of the status of each implementation in the series.
- A **static analyser**, which converts source-code into an intermediate representation (based on the abstract syntax tree) and calculates dependence information. This data feeds to the experiment manager and the program transformer (see below). [1]
- An **experiment manager**, which determines at each step the minimum set of experiments to be performed in order to gather the required run-time execution data, with help from the version manager, the static analyser and the user.
- A **database**, in which data from the static analyser and data collected during experiments is kept for each program version.
- A **performance analyser**, which analyses data from the database and produces summaries of the performance in terms of various overhead categories; the analysed data is presented to the user and passed to the program transformer.
- A **program transformer**, which supports the selection and, ultimately, automatic application of program transformations to improve parallel performance. [2]

Performance is analysed in terms of the extra execution time associated with a small number of categories of parallel overhead, including:

- **version cost:** the overhead of any serial changes made to a program so as to enable parallelism to be exploited;
- **unparallelised code cost:** the overhead associated with the serial fraction;
- **load imbalance cost:** due to uneven distribution of (parallel) work to processors;
- **memory access cost:** additional time spent in memory accesses, both within local cache hierarchies and to memory in remote processors;
- **scheduling cost:** for example, the overhead of parallel management code;
- **unexplained cost:** observed overhead not in one of the above classes.

The performance of each implementation is evaluated objectively, *relative to* either a "base" reference implementation (normally the best known *serial* implementation) or a previous version in the performance improvement history.

[1] Currently, FINESSE utilises the *Polaris* scanner [17] to support parsing of the source code and *Petit* [12] for dependence analysis.

[2] The prototype system does not implement automatic application of transformations; these are currently performed by hand.

Iteration of this process results in a semi-automatic, incremental development of implementations with progressively better performance (backtracking wherever it is found to be necessary).

2.1 Definitions

The performance of an implementation is evaluated in FINESSE by measuring its execution time on a range of multi-processor configurations. An ordered set P, where $P = \{p_1, p_2, \ldots, p_{max}\}$, and $\forall i, 1 \leq i < max, p_i < p_{i+1}$, is defined (in this paper, $p_1 = 1, p_2 = 2, \ldots, p_{max} = max$). Each parallel implementation is executed on $p = p_1, p_2, \ldots, p_{max}$ processors, and the overall execution time, T_p is measured for each execution. A *performance curve* for this code version executing on this target hardware platform is obtained by plotting $1/T_p$ versus p. The corresponding *performance vector* V is a p_{max}-tuple of performance values associated with execution using $p = 1, 2, 3, \ldots, p_{max}$ processors.

The *parallel effectiveness* ε_p of an execution with p processors is the ratio between the *actual performance gain* and the *ideal performance gain*, compared to the performance when $p = 1$. [3] Thus $\varepsilon_p = \frac{\frac{1}{T_p} - \frac{1}{T_1}}{\frac{p}{T_1} - \frac{1}{T_1}}$. This indicates how well the implementation utilises the *additional* processors (beyond $p = 1$) of a parallel hardware configuration. The value $\varepsilon_2 = 50\%$ means that when $p = 2$ only 50% of the second processor is being used; this is equivalent to a "speed-up" of 1.5, or an efficiency of 75%. The value $\varepsilon_p = 0\%$ means that the parallel implementation executes in exactly the same time as its serial counterpart. A negative value of ε_p means that parallel performance is worse than the serial counterpart. Note that ε_1 is always undefined since the ideal performance gain here equals zero.

The *parallel effectiveness vector* $\mathcal{E}_V$ is a p_{max}-tuple containing values of ε_p corresponding to each value of p. The *final parallel effectiveness* ($\varepsilon_{p_{max}}$) and the *average parallel effectiveness* (over the range $p = 2$ to $p = p_{max}$) are other useful measures which are used below; for the smoothly varying performance curves found in the experiments reported here, these two values serve to summarise the overall parallel effectiveness (the former is normally pessimistic, the latter optimistic over the range $p = 2, \ldots, p_{max}$).

3 Experimental Arrangement

The approach adopted in evaluating FINESSE is to parallelise six Fortran 77 test codes, using each of the parallelisation schemes described in Section 1, and to execute the resulting parallel codes on each of the two hardware platforms, a 4-processor SGI Challenge, known as *Ch/4*, and a 16-processor Origin 2000, *O2/16*. The set of test codes contains the following, well understood, application kernels:

[3] A reference serial execution is actually used for comparison, but this normally executes in time close to T_1.

- **Shallow–SH**: a simple atmospherics dynamics model.
- **Swim–SW**: another prototypical weather prediction code based on the shallow water equations.
- **Tred2 Eispack routine–T2**: a well know eigenvalue solver used for compiler benchmarking.
- **Molecular dynamics–MD**: an N-body simulation.
- **Airshed–AS**: a multiscale airshed simulation.
- **SPEC–SP**: a kernel for Legendre transforms used in weather prediction.

In this paper, details of the evaluation for **SP** are presented and results of the evaluation for all six codes are summarised. The evaluation of **T2** is described in more detail in [9]. Further details of the test codes and of the evaluation of FINESSE can be found in [8].

The **SP** code is described in [19]. It uses 3-dimensional arrays to represent vorticity, divergence, humidity and temperature. Each iteration of the main computational cycle repeatedly calls two subroutines to compute inverse and direct Legendre transforms, respectively, on a fixed number (in this case, 80) of *rows* along the latitudes. The loop-bounds for certain loops in the transformation routines are elements of two arrays which are initialised at run-time; these loops also make indirect array accesses. In serial execution, the initialisation routine accounts for only 5% of the total execution time; the two transformation routines share the remaining time equally.

4 Automatic versus Manual Parallelisation of SP

Completely manual and completely automatic parallelisations of **SP** are reported in [10]; these are conducted first by a human expert, User A (using the *Man* scheme), and then by the three automatic compilation systems introduced earlier. Quantitative performance data from these parallelisations forms the basis for comparison with later results obtained using FINESSE.

A performance curve is given for all versions of the test code executing on the two hardware platforms. Each figure showing performance curves also contains a line, labelled *Ideal*, which plots the *ideal parallel performance*, p/T_s. In each case, the corresponding elements of the performance vector, V, the parallel effectiveness vector, $\mathcal{E}_V$, and the *average parallel effectiveness*, $Avg(\varepsilon_p)$ (for $p = 2 \ldots p_{max}$), are shown in a table. Values of p_{max} are 4 for the *Ch/4* platform and 8 for the *O2/16* platform (operational constraints prevented testing with $p > 8$).

Figure 2 and Table 1 present these results for the parallelised **SP** test code on the *O2/16* platform (the tabular results for the *Ch/4* platform follow a similar pattern). On both platforms, the *Man* version performs better than any other because the main loops in two expensive and critical subroutines have been successfully parallelised. The final (average) parallel effectiveness of this version is 74% (99%) on the *O2/16* and 85% (86%) on the *Ch/4* platform. The *Pol* compiler is unable to parallelise *either* of the expensive loop nests. The remaining compilers, *PFA* and *Pfr*, detect parallelism in the *inner* loops of the expensive

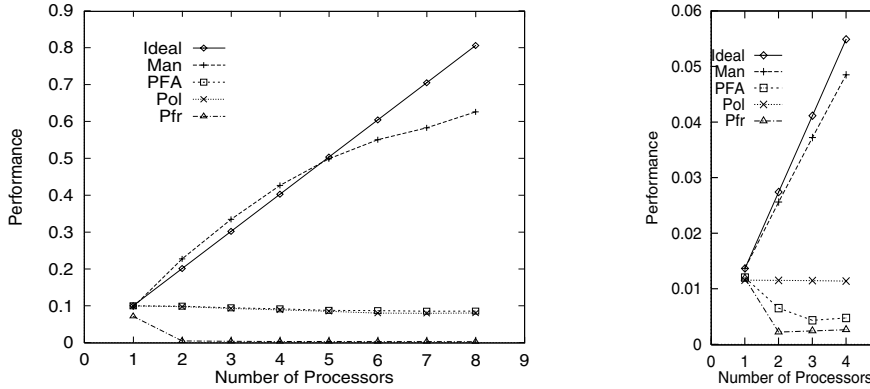


Fig. 2. Performance curves for **SP** code on the *O2/16* and *Ch/4* platforms.

loop nests. On the *Ch/4* platform, both schemes parallelise these inner loops, resulting in negative parallel effectiveness. On the *O2/16* platform, *PFA* refrains from parallelising these inner loops, but thereby merely inherits the serial performance (parallel effectiveness is everywhere close to 0%).

	p	2	3	4	5	6	7	8 final	$Avg(\varepsilon_p)$ $(p = 2, \ldots, 8)$
Man	V	0.228	0.335	0.427	0.499	0.551	0.583	0.627	
	$\mathcal{E}_V$	126%	116%	107%	99%	89%	80%	74%	99%
PFA	V	0.099	0.095	0.092	0.088	0.087	0.086	0.086	
	$\mathcal{E}_V$	-2%	-3%	-3%	-3%	-3%	-3%	-2%	-3%
Pol	V	0.098	0.093	0.089	0.085	0.081	0.080	0.081	
	$\mathcal{E}_V$	-3%	-4%	-4%	-4%	-4%	-4%	-4%	-4%
Pfr	V	0.005	0.004	0.003	0.003	0.003	0.003	0.003	
	$\mathcal{E}_V$	-95%	-48%	-32%	-24%	-19%	-16%	-14%	-36%

Table 1. Achieved performance of **SP** code on the *O2/16* platform.

5 Parallelisation of SP Using Finesse

This section describes the development, using Finesse, of parallel implementations of the **SP** test code on the *O2/16* platform. The efforts of an expert programmer (User A, once again) and of two other programmers (Users B and C — of varying high performance computing experience) are presented. For each user, program transformation, and thus creation of new implementations, is entirely guided by the overhead data produced by Finesse. For the most part,

the necessary tasks of the parallelisation are carried out automatically. User intervention is needed only at the time of program transformation, thus reducing programmer effort.

The initial environment for each development is set such that the *parallel profiling* experiments are run on $1, 2, 3$ and 4 processors, i.e. $P = \{1, 2, 3, 4\}$. Load imbalance overhead measurement experiments are conducted on $2, 3$ and 4 processors. Thus $p_{max} = 4$ in each case. Although the *O2/16* platform has up to 16 processors available, and results will be presented for values of p up to 8, the value $p_{max} = 4$ has been chosen here in order to reduce the number of executions for each version. Obviously, this hides performance effects for higher numbers of processors, and for practical use the initial environment of FINESSE would be set differently.

In order to compare the performance and parallel effectiveness of the FINESSE-generated codes with those of the corresponding *Man* versions, the finally accepted FINESSE version of the test code is executed on the Origin 2000 using $p_{max} = 8$.

Expert Parallelisation The parallel implementation of the **SP** code generated by User A using FINESSE shows significant improvement over the reference version. In [10] it is explained that exposing parallelism in the expensive loops of this code by application of available static analysis techniques is difficult. While parallelising using FINESSE, User A observed the execution behaviour of the code and then applied expert knowledge which made it possible to determine that the expensive loops can be parallelised. Hence User A re-coded these loops, causing Version 1 to be registered with, and analysed by, FINESSE; altogether three loops were parallelised. In order to analyse the performance of Version 1, 8 experiments were carried out by FINESSE; the analysis showed some parallel management overhead, some load imbalance overhead and some (negative) memory access and unexplained overhead. However, as the parallel effectiveness was close to that of the *Man* version, User A decided to stop at this point. [4] Table 2 provides a summary of the overhead data. Unfortunately it was not feasible to determine how much user-time was spent developing this implementation.

Figure 3 shows the performance curves for the *Man* version (from Section 4) and the final FINESSE version (Version 1). Table 3 presents the performance vectors (V), the parallel effectiveness vectors ($\mathcal{E}_V$) and the average parallel effectiveness of these two versions in the rows labelled *Man* and FINESSE, User-A.

Non-expert Parallelisation Two non-expert users (Users B and C) were also asked to parallelise the **SP** test code. User B spent approximately 2 hours, and managed to parallelise one of the two expensive loop nests, but was unable to expose parallelism in the other. User C spent about 8 hours, also parallelised one

[4] The problem of deciding when to stop the iterative development process is in general difficult, and the present implementation of FINESSE does not help with this. In the experiments reported here, the decision was made on arbitrary grounds; obviously User A was helped by specialist knowledge that would not usually be available.

Versions	p	ε_p	Overheads (in seconds)				
			O_V	O_{PM}	O_{UPC}	O_{LB}	$O_{MA\&Other}$
Version 0	-	-	unparallelised code overhead : 9.9593				
Version 1	1	-	0.0	0.5551	-	-	-0.0156
	2	121%	0.0	0.5551	0.0	0.0268	-1.0631
	3	114%	0.0	0.5551	0.0	0.0807	-0.9137
	4	107%	0.0	0.5551	0.0	0.0467	-0.7314

Table 2. **SP** code versions and associated overhead data (*O2/16* platform).

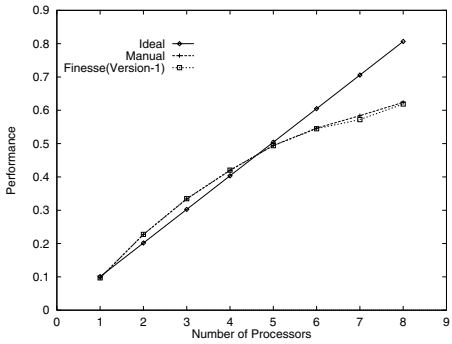


Fig. 3. Performance curves for **SP** on *O2/16* platform.

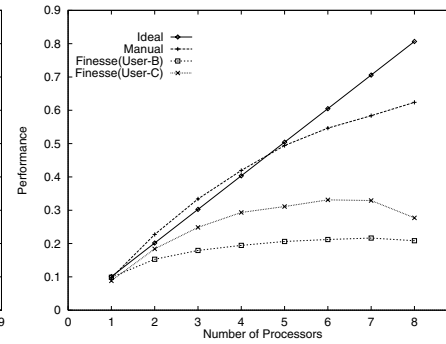


Fig. 4. Performance curves for **SP** (by Users B and C).

of the expensive loop nests and, in order to expose parallelism in the other, decided to apply array privatisation. This transformation requires expansion of all 3-dimensional arrays to the next higher dimension, thus incurring a large amount of memory access overhead. Hence, performance of the transformed version did not improve as desired.

Figure 4 depicts the pertinent performance curves. Table 3 compares the performance vectors (V), the parallel effectiveness vectors ($\mathcal{E}_V$) and the average parallel effectiveness of these two versions (labelled FINESSE, User-B and User-C) with those of the *Man* version (from Section 4).

6 Summary of Results for All Six Test Codes

The expert, User A, parallelised all six test codes with assistance from FINESSE; clearly, experience obtained earlier, when studying fully manual and fully automatic parallelisations, must have made it easier to achieve good results. Two further non-expert users (Users D and E) also undertook FINESSE-assisted parallelisation of one test code each. Thus, a total of ten FINESSE-assisted parallelisations were conducted altogether. Table 4 compares the final parallel effectiveness of all ten generated codes with that of the manually parallelised codes and the best of the compiler-generated codes.

	p	2	3	4	5	6	7	8 final	$Avg(\varepsilon_p)$ $(p = 2, \ldots, 8)$
Man	V	0.227	0.334	0.419	0.494	0.546	0.584	0.624	
	$\mathcal{E}_V$	125%	116%	105%	98%	88%	80%	74%	98%
FINESSE	V	0.227	0.334	0.420	0.494	0.545	0.572	0.619	
(User-A)	$\mathcal{E}_V$	126%	115%	106%	97%	88%	78%	73%	98%
FINESSE	V	0.153	0.179	0.195	0.206	0.212	0.217	0.208	
(User-B)	$\mathcal{E}_V$	52%	39%	31%	26%	22%	19%	15%	29%
FINESSE	V	0.184	0.248	0.293	0.311	0.331	0.329	0.277	
(User-C)	$\mathcal{E}_V$	83%	73%	64%	52%	46%	38%	25%	54%

Table 3. Achieved performance of **SP** code (User B and C) on *O2/16* platform.

code	**SH**	**SW**	**T2**		**MD**		**AS**	**SP**		
User	A	A	A	D	A	E	A	A	B	C
FINESSE	265%	85%	40%	39%	85%	75%	88%	74%	15%	25%
Manual	277%	122%	40%		85%		93%	73%		
Best-compiler	261%	87%	-6%		25%		8%	-2%		

Table 4. Final parallel effectiveness compared for FINESSE, *Man* and the best of the compilers.

Overall, the results obtained using FINESSE compare favourably with those obtained via expert manual parallelisation. Moreover, in many of these cases, use of autoparallelising compilers does not produce good results.

7 Related Work

The large majority of existing tools which support the parallelisation process are aimed at the message-passing programming paradigm. Space does not permit an exhaustive list of these tools, but, restricting attention to tools which support FORTRAN or C, significant research efforts in this area include Paragraph [3], Pablo [15] and Paradyn [7]. Commercial systems include Vampir [11] and MPP-Apprentice [21].

Tools for shared memory systems have received less attention, possibly due to the lack of a widely accepted standard for the associated programming paradigm, and because of the need for hardware support to monitor the memory system (the advent of OpenMP [13] seems likely to ease the former situation, while the PerfAPI [2] standards initiative will ameliorate the latter).

SUIF Explorer [5] is a recent tool which also integrates the user directly into the parallelisation process. The user is guided towards the diagnosis of performance problems by a *Performance GURU* and affects performance by, for example, placing assertions in the source code which assist the compiler in parallelising the code. SUIF explorer uses the concept of *program slicing* [20] to focus the user's attention on the relevant parts of the source code.

Carnival [6] supports *waiting time analysis* for both message-passing and shared memory systems. This is the nearest to full overheads profiling, as required by FINESSE, that any other tool achieves; however, important overhead categories, such as memory accesses, are excluded, and no reference code is used to give an unbiased basis for comparison. In the Blizzard software VSM system, Paradyn [7] has been adapted to monitor memory accesses and memory-sharing behaviour. SVMview [1] is another tool designed for a software VSM system (FORTRAN-S); however techniques for monitoring software systems do not transfer readily to hardware-supported VSM.

Commercial systems include ATExpert [4] (a precursor to MPP-Apprentice), for Cray vector SMPs, and Codevision, for Silicon Graphics shared memory machines, including the Origin 2000. Codevision is a profiling tool, allowing profiles of not only the program counter, but also the hardware performance counters available on the Origin 2000. It also features an experiment management framework.

8 Conclusion

The results presented in Sections 5 and 6 provide limited evidence that use of a tool such as FINESSE enables a User to improve the (parallel) performance of a given program in a relatively short time (and, hence, at relatively low cost). In most cases, the parallel codes generated using FINESSE perform close to the corresponding versions developed entirely manually. The results described in Sections 4 and 6 demonstrate that users, with no prior knowledge of the codes, can effectively use this tool to produce acceptable parallel implementations quickly, compared to the manual method, simply due to its experiment management support. Comments obtained from the users showed that, while using FINESSE, they spent most of their time selecting suitable transformations and then implementing them. This time could be further reduced by implementing (firstly) an automatic program transformer, and (more ambitiously) a tranformation recommender.

In some cases, and particularly in the case of the **SP** code, users are not really successful at producing a high performance parallel implementation. Efficient parallelisation of this code is possible only by careful analysis of the code structure, as well as its execution behaviour. It is believed that, if the static analyser and the performance analyser of the prototype tool were to be further improved, then users could generate more efficient parallel implementations of this code. In any case, none of the automatic compilers is able to improve the performance of this code (indeed, they often worsen it); in contrast, each FINESSE-assisted user was able to improve performance, albeit by a limited amount.

References

1. D. Badouel, T. Priol and L. Renambot, *SVMview: a Performance Tuning Tool for DSM-based Parallel Computers*, Tech. Rep. PI-966, IRISA Rennes, Nov. 1995.

2. J. Dongarra, *Performance Data Standard and API*, available at: http://www.cs.utk.edu/ mucci/pdsa/perfapi/index.htm
3. I. Glendinning, V.S. Getov, A. Hellberg, R.W. Hockney and D.J. Pritchard, *Performance Visualisation in a Portable Parallel Programming Environment*, Proc. Workshop on Monitoring and Visualization of Parallel Processing Systems, Moravany, CSFR, Oct. 1992.
4. J. Kohn and W. Williams, *ATExpert*, J. Par. and Dist. Comp. **18**, 205–222, 1993.
5. S-W Liao, A. Diwan, R.P. Bosch, A. Ghuloum and M.S Lam, *SUIF Explorer: An Interactive and Interprocedural Parallelizer*, ACM SIGPLAN Notices **34**(8), 37–48, 1999.
6. W. Meira Jr., T.J. LeBlanc and A. Poulos, *Waiting Time Analysis and Performance Visualization in Carnival*, ACM SIGMETRICS Symp. on Par. and Dist. Tools, 1–10, May 1996.
7. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam and T. Newhall, *The Paradyn Parallel Performance Measurement Tools*, IEEE Computer **28**(11), 37–46, 1995.
8. N. Mukherjee, On the Effectiveness of Feedback-Guided Parallelisation, PhD thesis, Univ. Manchester, Sept. 1999.
9. N. Mukherjee, G.D. Riley and J.R. Gurd, FINESSE*: A Prototype Feedback-guided Performance Enhancement System*, Proc. 8th Euromicro Workshop on Parallel and Distributed Processing, 101–109, Jan. 2000.
10. N. Mukherjee and J.R. Gurd, *A comparative analysis of four parallelisation schemes*, Proc. ACM Intl. Conf. Supercomp., 278–285, June 1999.
11. W.E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, available at: http://www.kfa-juelich.de/zam/PTdocs/vampir/vampir.html
12. The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs, available at: http:// www.cs.umd.edu/projects/omega
13. OpenMP Architecture Review Board, *OpenMP* FORTRAN *Application Interface*, available at: http://www.openmp.org/openmp/ mp-documents/fspec.A4.ps
14. Parafrase-2, A Vectorizing/Parallelizing Compiler, available at: http://www.csrd.uiuc.edu/parafrase2
15. D.A. Reed, *Experimental Analysis of Parallel Systems: Techniques and Open Problems*, Lect. Notes in Comp. Sci. **794**, 25–51, 1994.
16. *POWER* FORTRAN *Accelerator User's Guide.*
17. Polaris, Automatic Parallelization of Conventional FORTRAN Programs, available at: http://polaris.cs.uiuc.edu/polaris/ polaris.html
18. G.D. Riley, J.M. Bull and J.R. Gurd, *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics*, Proc. ACM Intl. Conf. Supercomp., 36–43, July 1997.
19. D. F. Snelling, *A High Resolution Parallel Legendre Transform Algorithm*, Proc. ACM Intl. Conf. Supercomp., Lect. Notes in Comp. Sci. **297**, pp. 854–862, 1987.
20. M. Weiser, *Program Slicing*, IEEE Trans. Soft. Eng., **10**(4), 352–357, 1984.
21. W. Williams, T. Hoel and D. Pase, *The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D*, **in**: K.M. Decker et al. (eds.), *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, 333–345, 1994.

On Combining Computational Differentiation and Toolkits for Parallel Scientific Computing*

Christian H. Bischof[1], H. Martin Bücker[1], and Paul D. Hovland[2]

[1] Institute for Scientific Computing, Aachen University of Technology,
52056 Aachen, Germany,
{bischof,buecker}@sc.rwth-aachen.de,
http://www.sc.rwth-aachen.de

[2] Mathematics and Computer Science Division, Argonne National Laboratory,
9700 South Cass Ave, Argonne, IL 60439, USA,
hovland@mcs.anl.gov,
http://www.mcs.anl.gov

Abstract. Automatic differentiation is a powerful technique for evaluating derivatives of functions given in the form of a high-level programming language such as Fortran, C, or C++. The program is treated as a potentially very long sequence of elementary statements to which the chain rule of differential calculus is applied over and over again. Combining automatic differentiation and the organizational structure of toolkits for parallel scientific computing provides a mechanism for evaluating derivatives by exploiting mathematical insight on a higher level. In these toolkits, algorithmic structures such as BLAS-like operations, linear and nonlinear solvers, or integrators for ordinary differential equations can be identified by their standardized interfaces and recognized as high-level mathematical objects rather than as a sequence of elementary statements. In this note, the differentiation of a linear solver with respect to some parameter vector is taken as an example. Mathematical insight is used to reformulate this problem into the solution of multiple linear systems that share the same coefficient matrix but differ in their right-hand sides. The experiments reported here use ADIC, a tool for the automatic differentiation of C programs, and PETSc, an object-oriented toolkit for the parallel solution of scientific problems modeled by partial differential equations.

1 Numerical versus Automatic Differentiation

Gradient methods for optimization problems and Newton's method for the solution of nonlinear systems are only two examples showing that computational techniques require the evaluation of derivatives of some objective function. In large-scale scientific applications, the objective function $f : \mathbb{R}^n \to \mathbb{R}^m$ is typically not available in analytic form but is given by a computer code written in a

* This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 86–94, 2000.
© Springer-Verlag Berlin Heidelberg 2000

high-level programming language such as Fortran, C, or C++. Think of f as the function computed by, say, a (parallel) computational fluid dynamics code consisting of hundreds of thousands lines that simulates the flow around a complex three-dimensional geometry. Given such a representation of the objective function $f(\mathbf{x}) = \big(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})\big)^T$, computational methods often demand the evaluation of the Jacobian matrix

$$J_f(\mathbf{x}) := \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_m(\mathbf{x}) & \ldots & \frac{\partial}{\partial x_n} f_m(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

at some point of interest $\mathbf{x} \in \mathbb{R}^n$.

Deriving an analytic expression for $J_f(\mathbf{x})$ is often inadequate. Moreover, implementing such an analytic expression by hand is challenging, error-prone, and time-consuming. Hence, other approaches are typically preferred.

A well-known and widely used approach for the approximation of the Jacobian matrix is divided differences (DD). For the sake of simplicity, we mention only first-order forward DD but stress that the following discussion applies to DD as a technique of numerical differentiation in general. Using first-order forward DD, one can approximate the ith column of the Jacobian matrix (1) by

$$\frac{f(\mathbf{x} + h_i \mathbf{e}_i) - f(\mathbf{x})}{h_i}, \tag{2}$$

where h_i is a suitably-chosen step size and $\mathbf{e}_i \in \mathbb{R}^n$ is the ith Cartesian unit vector. An advantage of the DD approach is that the function f need be evaluated only at some suitably chosen points. Roughly speaking, f is used as a black box evaluated at some points. The main disadvantage of DD is that the accuracy of the approximation depends crucially on a suitable choice of these points, specifically, of the step size h_i. There is always the dilemma that the step size should be small in order to decrease the truncation error of (2) and that, on the other hand, the step size should be large to avoid cancellation errors using finite-precision arithmetic when evaluating (2).

Analytic and numerical differentiation methods are often considered to be the only options for computing derivatives. Another option, however, is symbolic differentiation by computer algebra packages such as Macsyma or Mathematica. Unfortunately, because of the rapid growth of the underlying explicit expressions for the derivatives, traditional symbolic differentiation is currently inefficient [9].

Another technique for computing derivatives of an objective function is automatic differentiation (AD) [10, 16]. Given a computer code for the objective function in virtually any high-level programming language such as Fortran, C, or C++, automatic differentiation tools such as ADIFOR [4, 5], ADIC [6], or ADOL-C [13] can by applied in a black-box fashion. A survey of AD tools can be found at `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools`. These tools generate another computer program, called a derivative-enhanced program, that evaluates $f(\mathbf{x})$ and $J_f(\mathbf{x})$ simultaneously. The key concept behind AD is

the fact that every computation, no matter how complicated, is executed on a computer as a—potentially very long—sequence of a limited set of elementary arithmetic operations such as additions, multiplications, and intrinsic functions such as `sin()` and `cos()`. By applying the chain rule of differential calculus over and over again to the composition of those elementary operations, $f(\mathbf{x})$ and $J_f(\mathbf{x})$ can be evaluated up to machine precision. Besides the advantage of accuracy, AD requires minimal human effort and has been proven more efficient than DD under a wide range of circumstances [3, 5, 12].

2 Computational Differentiation in Scientific Toolkits

Given the fact that automatic differentiation tools need not know anything about the underlying problem whose code is being differentiated, the resulting efficiency of the automatically generated code is remarkable. However, it is possible not only to apply AD technology in a black-box fashion but also to couple the application of AD with high-level knowledge about the underlying code. We refer to this combination as *computational differentiation* (CD). In some cases, CD can reduce memory requirements, improve performance, and increase accuracy. For instance, a CD strategy identifying a major computational component, deriving its analytical expression, and coding the corresponding derivatives by hand is likely to perform better than the standard AD approach that can operate only on the level of simple arithmetic operations.

In toolkits for scientific computations, algorithmic structures can be automatically recognized when applying AD tools, provided standardized interfaces are available. Examples include standard (BLAS-like) linear algebra kernels, linear and nonlinear solvers, and integrators for ordinary differential equations. These algorithmic structures are the key to exploiting high-level knowledge when CD is used to differentiate applications written in toolkits such as the Portable, Extensible Toolkit for Scientific Computation (PETSc) [1, 2].

Consider the case of differentiating a code for the solution of sparse systems of linear equations. PETSc provides a uniform interface to a variety of methods for solving these systems in parallel. Rather than applying an AD tool in a black-box fashion to a particular method as a sequence of elementary arithmetic operations, the combination of CD and PETSc allows us to generate a single derivative-enhanced program for any linear solver. More precisely, assume that we are concerned with a code for the solution of

$$A \cdot \mathbf{x}(\mathbf{s}) = \mathbf{b}(\mathbf{s}) \tag{3}$$

where $A \in \mathbb{R}^{N \times N}$ is the coefficient matrix. For the sake of simplicity, it is assumed that only the solution $\mathbf{x}(\mathbf{s}) \in \mathbb{R}^N$ and the right-hand side $\mathbf{b}(\mathbf{s}) \in \mathbb{R}^N$, but not the coefficient matrix, depend on a free parameter vector $\mathbf{s} \in \mathbb{R}^r$. The code for the solution of (3) implicitly defines a function $\mathbf{x}(\mathbf{s})$. Now, suppose that one is interested in the Jacobian $J_{\mathbf{x}}(\mathbf{s}) \in \mathbb{R}^{N \times r}$ of the solution $\mathbf{x}$ with respect to the free parameter vector $\mathbf{s}$. Differentiating (3) with respect to $\mathbf{s}$ yields

$$A \cdot J_{\mathbf{x}}(\mathbf{s}) = J_{\mathbf{b}}(\mathbf{s}), \tag{4}$$

where $J_{\mathbf{b}}(\mathbf{s}) \in \mathbb{R}^{N \times r}$ denotes the Jacobian of the right-hand side $\mathbf{b}$.

In parallel high-performance computing, the coefficient matrix A is often large and sparse. For instance, numerical simulations based on partial differential equations typically lead to such systems. Krylov subspace methods [17] are currently considered to be among the most powerful techniques for the solution of sparse linear systems. These iterative methods generate a sequence of approximations to the exact solution $\mathbf{x}(\mathbf{s})$ of the system (3). Hence, an implementation of a Krylov subspace method does not compute the function $\mathbf{x}(\mathbf{s})$ but only an approximation to that function. Since, in this case, AD is applied to the approximation of a function rather than to the function itself, one may ask whether and how AD-produced derivatives are reasonable approximations to the desired derivatives of the function $\mathbf{x}(\mathbf{s})$. This sometimes undesired side-effect is discussed in more detail in [8, 11] and can be minimized by the following CD approach.

Recall that the standard AD approach would process the given code for a particular linear solver for (3), say an implementation of the biconjugate gradient method, as a sequence of binary additions, multiplications, and the like. In contrast, combining the CD approach with PETSc consists of the following steps:

1. Recognize from inspection of PETSc's interface that the code is meant to solve a linear system of type (3) regardless of which particular iterative method is used.
2. Exploit the knowledge that the Jacobian $J_{\mathbf{x}}(\mathbf{s})$ is given by the solution of the multiple linear systems (4) involving the same coefficient matrix, but r different right-hand sides.

The CD approach obviously eliminates the above mentioned problems with automatic differentiation of iterative schemes for the approximation of functions. There is also the advantage that the CD approach abstracts from the particular linear solver. Differentiation of codes involving any linear solver, not only those making use of the biconjugate gradient method, benefits from an efficient technique to solve (4).

3 Potential Gain of CD and Future Research Directions

A previous study [14] differentiating PETSc with ADIC has shown that, for iterative linear solvers, CD-produced derivatives are to be preferred to derivatives obtained from AD or DD. More precisely, the findings from that study with respect to differentiation of linear solvers are as follows. The derivatives produced by the CD and AD approaches are several orders of magnitude more accurate than those produced by DD. Compared with AD, the accuracy of CD is higher. In addition, the CD-produced derivatives are obtained in less execution time than those by AD, which in turn is faster than DD. The differences in execution time between these three approaches increase with increasing the dimension, r, of the free parameter vector $\mathbf{s}$.

While the CD approach turns out to be clearly the best of the three discussed approaches, its performance can be improved significantly. The linear systems (4)

involving the same coefficient matrix but r different right-hand sides are solved in [14] by running r times a typical Krylov subspace method for a linear system with a single right-hand side. In contrast to these successive runs, so-called block versions of Krylov subspace methods are suitable candidates for solving systems with multiple right-hand sides; see [7, 15] and the references given there. In each block iteration, block Krylov methods generate r iterates simultaneously, each of which is designed to be an approximation to the exact solutions of a single system. Note that direct methods such as Gaussian elimination can be trivially adapted to multiple linear systems because their computational work is dominated by the factorization of the coefficient matrix. Once the factorization is available, the solutions of multiple linear systems are given by a forward and back substitution per right-hand side. However, because of the excessive amount of fill-in, direct methods are often inappropriate for large sparse systems.

In this note, we extend the work reported in [14] by incorporating iterative block methods into the CD approach. Based on the given scenario of the combination of the ADIC tool and the PETSc package, we consider a parallel implementation of a block version of the biconjugate gradient method [15]. We focus here on some fundamental issues illustrating this approach; a rigorous numerical treatment will be presented elsewhere. To demonstrate the potential gain from using a block method in contrast to successive runs of a typical iterative method, we take the number of matrix-vector multiplications as a rough performance measure. This is a legitimate choice because, usually, the matrix-vector multiplications dominate the computational work of an iterative method for large sparse systems.

Figure 1 shows, on a log scale, the convergence behavior of the block biconjugate gradient method applied to a system arising from a discretization of a two-dimensional partial differential equation of order $N = 1,600$ with $r = 3$ right-hand sides. Throughout this note, we always consider the relative residual norm, that is, the Euclidean norm of the residual scaled by the Euclidean norm of the initial residual. In this example, the iterates for the $r = 3$ systems converge at the same step of the block iteration. In general, however, these iterates converge at different steps. Future work will therefore be concerned with how to detect and deflate converged systems. Such deflation techniques are crucial to block methods because the algorithm would break down in the next block iteration step; see the discussion in [7] for more details on deflation. We further assume that block iterates converge at the same step and that deflation is not necessary.

Next, we consider a finer discretization of the same equation leading to a larger system of order $N = 62,500$ with $r = 7$ right-hand sides to illustrate the potential gain of block methods. Figure 2 compares the convergence history of applying a block method to obtain block iterates for all $r = 7$ systems simultaneously and running a typical iterative method for a single right-hand side $r = 7$ times one after another. For all our experiments, we use the biconjugate gradient method provided by the linear equation solver (SLES) component of PETSc as a typical iterative method for a single right-hand side. For the plot

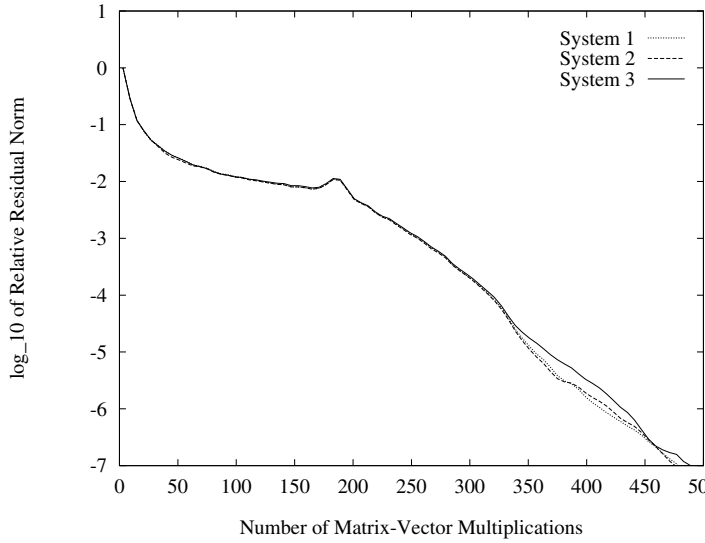


Fig. 1. Convergence history of the block method for the solution of $r = 3$ systems involving the same coefficient matrix of order $N = 1,600$. The residual norm is shown for each of the systems individually.

of the block method we use the largest relative residual norm of all systems. In this example, the biconjugate gradient method for a single right-hand side (dotted curve) needs $8,031$ matrix-vector multiplications to achieve a tolerance of 10^{-7} in the relative residual norm. The block method (solid curve), on the contrary, converges in only $5,089$ matrix-vector multiplications to achieve the same tolerance. Clearly, block methods offer a potential speedup in comparison with successive runs of methods for a single right-hand side.

The ratio of the number of matrix-vector multiplications of the method for a single right-hand side to the number of matrix-vector multiplications of the block method is 1.58 in the example above and is given in the corresponding column of Table 1. In addition to the case where the number of right-hand sides is $r = 7$, this table contains the results for the same coefficient matrix, but for varying numbers of right-hand sides. It is not surprising that the number of matrix-vector multiplications needed to converge increases with an increasing number of right-hand sides r. Note, however, that the ratio also increases with r. This behavior shows that the larger the number of right-hand sides the more attractive the use of block methods.

Many interesting aspects remain to be investigated. Besides the above mentioned deflation technique, there is the question of determining a suitable preconditioner. Here, we completely omitted preconditioning in order to make the comparison between the block method and its correspondence for a single right-hand side more visible. Nevertheless, preconditioning is an important ingredient in any iterative technique for the solution of sparse linear systems for both single

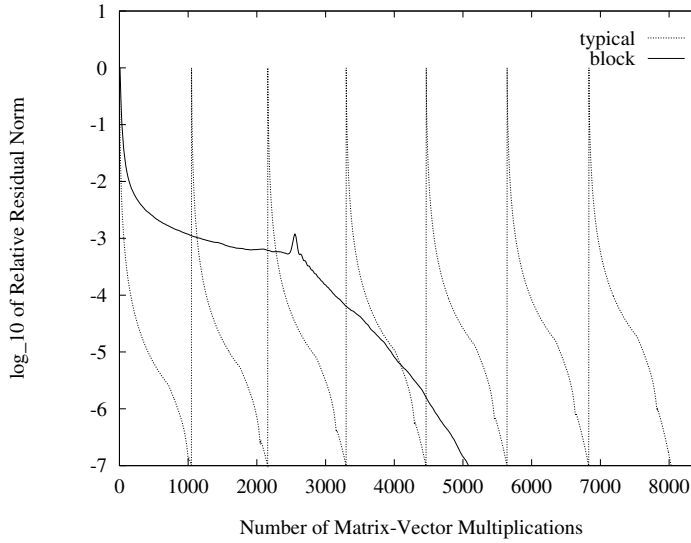


Fig. 2. Comparison of the block method for the solution of $r = 7$ systems involving the same coefficient matrix of order $N = 62,500$ and r successive runs of a typical iterative method for a single right-hand side.

and multiple right-hand sides. Notice that, in their method, Freund and Malhotra [7] report a dependence of the choice of an appropriate preconditioner on the parameter r.

Block methods are also of interest because they offer the potential for better performance. At the single-processor level, performing several matrix-vector products simultaneously provides increased temporal locality for the matrix, thus mitigating the effects of the memory bandwidth bottleneck. The availability of several vectors at the same time also provides opportunities for increased

Table 1. Comparison of matrix-vector multiplications needed to require a decrease of seven orders of magnitude in the relative residual norm for different dimensions, r, of the free parameter vector. The rows show the number of matrix-vector multiplications for r successive runs of a typical iterative method for a single right-hand side, a corresponding block version, and their ratio, respectively. (The order of the matrix is $N = 62,500$.)

r	1	2	3	4	5	6	7	8	9	10
typical	1,047	2,157	3,299	4,463	5,641	6,831	8,031	9,237	10,451	11,669
block	971	1,770	2,361	3,060	3,815	4,554	5,089	5,624	6,219	6,550
ratio	1.08	1.22	1.40	1.46	1.48	1.50	1.58	1.64	1.68	1.78

parallel performance, as increased data locality reduces the ratio of communication to computation. Even for the single right-hand side case, block methods are attractive because of their potential for exploiting locality, a key issue in implementing techniques for high-performance computers.

4 Concluding Remarks

Automatic differentiation applied to toolkits for parallel scientific computing such as PETSc increases their functionality significantly. While automatic differentiation is more accurate and, under a wide range of circumstances, faster than approximating derivatives numerically, its performance can be improved even further by exploiting high-level mathematical knowledge. The organizational structure of toolkits provides this information in a natural way by relying on standardized interfaces for high-level algorithmic structures. The reason why improvements over the traditional form of automatic differentiation are possible is that, in the traditional approach, any program is treated as a sequence of elementary statements. Though powerful, automatic differentiation operates on the level of statements. In contrast, *computational differentiation*, the combination of mechanically applying techniques of automatic differentiation and human-guided mathematical insight, allows the analysis of objects on higher levels than on the level of elementary statements. These issues are demonstrated by taking the differentiation of an iterative solver for the solution of large sparse systems of linear equations as an example. Here, mathematical insight consists in reformulating the differentiation of a linear solver into a solution of multiple linear systems involving the same coefficient matrix, but whose right-hand sides differ. The reformulation enables the integration of appropriate techniques for the problem of solving multiple linear systems, leading to a significant performance improvement when differentiating code for any linear solver.

Acknowledgments

This work was completed while the second author was visiting the Mathematics and Computer Science Division, Argonne National Laboratory. He was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. Gail Pieper proofread a draft of this manuscript and gave several helpful comments.

References

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.24, Argonne National Laboratory, 1999.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[3] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, 1996.

[4] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[5] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[6] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[7] Roland W. Freund and Manish Malhotra. A block QMR algorithm for non-Hermitian linear systems with multiple right-hand sides. *Linear Algebra and Its Applications*, 254:119–157, 1997.

[8] Jean-Charles Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13–22, 1992.

[9] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[10] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[11] Andreas Griewank, Christian Bischof, George Corliss, Alan Carle, and Karen Williamson. Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[12] Andreas Griewank and George Corliss. *Automatic Differentiation of Algorithms.* SIAM, Philadelphia, 1991.

[13] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[14] Paul Hovland, Boyana Norris, Lucas Roh, and Barry Smith. Developing a derivative-enhanced object-oriented toolkit for scientific computations. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 129–137, Philadelphia, 1999. SIAM.

[15] Dianne P. O'Leary. The block conjugated gradient algorithm and related methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.

[16] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[17] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, 1996.

Generating Parallel Program Frameworks from Parallel Design Patterns

Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steven Bromling

Department of Computing Science, University of Alberta, CANADA T6G 2H1
{stevem,duane,jonathan,bromling}@cs.ualberta.ca

Abstract. Object-oriented programming, design patterns, and frameworks are abstraction techniques that have been used to reduce the complexity of sequential programming. The CO_2P_3S parallel programming system provides a layered development process that applies these three techniques to the more difficult domain of parallel programming. The system generates correct frameworks from pattern template specifications at the highest layer and provides performance tuning opportunities at lower layers. Each of these features is a solution to a major problem with current parallel programming systems. This paper describes CO_2P_3S and its highest level of abstraction using an example program to demonstrate the programming model and one of the supported pattern templates. Our results show that a programmer using the system can quickly generate a correct parallel structure. Further, applications built using these structures provide good speedups for a small amount of development effort.

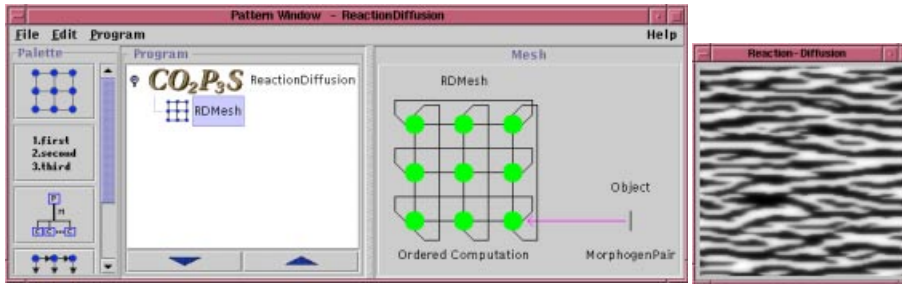
1 Introduction

Parallel programming offers substantial performance improvements to computationally intensive problems from fields such as computational biology, physics, chemistry, and computer graphics. Some of these problems require hours, days, or even weeks of computing time. However, using multiple processors effectively requires the creation of highly concurrent algorithms. These algorithms must then be implemented correctly and efficiently. This task is difficult, and usually falls on a small number of experts.

To simplify this task, we turn to abstraction techniques and development tools. From sequential programming, we note that the use of abstraction techniques such as object-oriented programming, design patterns, and frameworks reduces the software development effort. Object–oriented programming has proven successful through techniques such as encapsulation and code reuse. Design patterns document solutions to recurring design problems that can be applied in a variety of contexts [1]. Frameworks provide a set of classes that implement the basic structure of a particular kind of application, which are composed and specialized by a programmer to quickly create complete applications [2]. A development tool, such as a parallel programming system, can provide a complete toolset to support the development, debugging, and performance tuning stages of parallel programming.

The CO_2P_3S parallel programming system (Correct Object-Oriented Pattern-based Parallel Programming System, or "cops") combines the three abstraction techniques using a layered programming model that supports both the fast development of parallel

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 95–104, 2000.
© Springer-Verlag Berlin Heidelberg 2000



(a) A screenshot of the Mesh template in CO_2P_3S. (b) Output image.

Fig. 1. The reaction–diffusion example in CO_2P_3S with an example texture.

programs and the ability to tune the resulting programs for performance [3,4]. The highest level of abstraction in CO_2P_3S emphasizes correctness by generating parallel structural code for an application based on a pattern description of the structure. The lower layers emphasize openness [7], gradually exposing the implementation details of the generated code to introduce opportunities for performance debugging. Users can select an appropriate layer of abstraction based on their needs.

This approach advances the state of the art in pattern-based parallel programming systems research by providing a solution to two recurring problems. First, CO_2P_3S generates correct parallel structural code for the user based on a pattern description of the program. In contrast, current pattern-based systems also require a pattern description but then rely on the user to provide application code that matches the selected structure. Second, the openness provided by the lower layers of CO_2P_3S gives the user the ability to tune an application in a structured way. Most systems restrict the user to the provided programming model and provide no facility for performance improvements. Those systems that do provide openness typically strip away all abstractions in the programming model immediately, overwhelming the user with details of the run-time system.

CO_2P_3S provides three layers of abstraction: the Patterns Layer, the Intermediate Code Layer, and the Native Code Layer. The Patterns Layer supports pattern-based parallel program development through framework generation. The user expresses the concurrency in a program by manipulating graphical representations of parallel design pattern templates. A template is a design pattern that is customized for the application via template parameters supplied by the user through the user interface. From the pattern specification, CO_2P_3S generates a framework implementing the selected parallel structure. The user fills in the application–dependent parts of the framework to implement a program. The two remaining layers, the Intermediate Code Layer and the Native Code Layer, allow users to modify the structure and implementation of the generated framework for performance tuning. More details on CO_2P_3S can be found in [3,4].

In this paper, we highlight the development model and user interface of CO_2P_3S using an example problem. CO_2P_3S is implemented in Java and creates multithreaded parallel frameworks that execute on shared memory systems using a native–threaded JVM that allows threads to be mapped to different processors. Our example is a reaction–

diffusion texture generation program that performs a chemical simulation to generate images resembling zebra stripes, shown in Figure 1. This program uses a Mesh pattern template which is an example of a parallel structural pattern for the SPMD model. We discuss the development process and the performance of this program. We also briefly discuss two other patterns supported by CO_2P_3S and another example problem. Our results show that the Patterns Layer is capable of quickly producing parallel programs that obtain performance gains.

2 Reaction–Diffusion Texture Generation

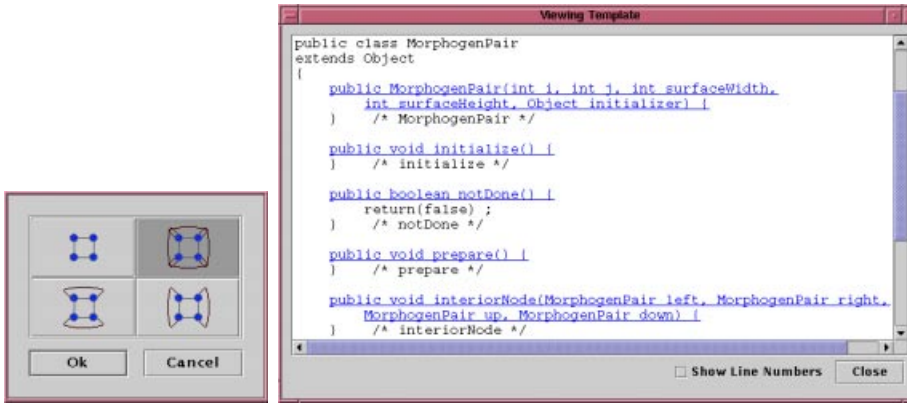
This section describes an example program that uses one of the CO_2P_3S pattern templates. The goal is to show how CO_2P_3S simplifies the task of parallel programming by generating correct framework code from a pattern template. This allows a user to write only sequential code to implement a parallel program. To accomplish this goal, a considerable amount of detail is given about the pattern template, its parameters, and the framework code that is generated.

Reaction–diffusion texture generation simulates two chemicals called *morphogens* as they simultaneously diffuse over a two–dimensional surface and react with one another [8]. This simulation, starting with random concentrations of each morphogen across the surface, can produce texture maps that approximate zebra stripes, as shown in Figure 1(b). This problem is solved using convolution. The simulation executes until the change in concentration for both morphogens at every point on the surface falls below a threshold. This implementation allows the diffusion of the morphogens to wrap around the edges of the surface. The resulting texture map can be tiled on a larger display without any noticeable edges between tiles.

2.1 Design Pattern Selection

The first step in implementing a pattern-based program is to analyze the problem and select the appropriate set of design patterns. This process still represents the bottleneck in the design of any program. We do not address pattern selection in this paper, but one approach is discussed in [5]. Given our problem, the two–dimensional Mesh pattern is a good choice. The problem is an iterative algorithm executed over the elements of a two–dimensional surface. The concentration of an element depends only on its current concentration and the current concentrations of its neighbouring elements. These computations can be done concurrently, as long as each element waits for its neighbours to finish before continuing with its next iteration.

Figure 1(a) shows a view of the reaction–diffusion program in CO_2P_3S. The user has selected the Mesh pattern template from the palette and has provided additional information (via dialog boxes such as that in Figure 2(a)) to specify the parameters for the template. The Mesh template requires the class name for the mesh object and the name of the mesh element class. The mesh object is responsible for properly executing the mesh computation, which the user defines by supplying the implementation of hook methods for the mesh element class. For this application, the user has indicated that the mesh object should be an instance of the `RDMesh` class and the mesh elements are



(a) Boundary conditions for the Mesh.

(b) Viewing template, showing default implementations of hook methods.

Fig. 2. Two dialogs from CO_2P_3S.

instances of the `MorphogenPair` class. The user has also specified that the `MorphogenPair` class has no user–defined superclass, so the class `Object` is used.

In addition to the class names, the user can also define parameters that affect the mesh computation itself. This example problem requires a fully–toroidal mesh, so the edges of the surface wrap around. The mesh computation considers the neighbours of mesh elements on the edges of the surface to be elements on the opposite edge, implementing the required topology. The user has selected this topology from the dialog in Figure 2(a), which also provides vertical–toroidal, horizontal–toroidal, and non–toroidal options for the topology. Further, this application requires an instance of the Mesh template that uses a four–point mesh, using the neighbours on the four compass points, as the morphogens diffuse horizontally and vertically. Alternatively, the user can select an eight–point mesh for problems that require data from all eight neighbouring elements. Finally, the new value for a morphogen in a given iteration is based on values computed in the previous iteration. Thus, the user must select an *ordered* mesh, which ensures that iterations are performed in lock step for all mesh elements. Alternatively, the user can select a *chaotic* mesh, where an element proceeds with its next iteration as soon as it can rather than waiting for its neighbours to finish. All of these options are available in the Mesh Pattern template through the CO_2P_3S user interface in Figure 1(a).

2.2 Generating and Using the Mesh Framework

Once the user has specified the parameters for the Mesh pattern template, CO_2P_3S uses the template to generate a framework of code implementing the structure for that specific version of the Mesh. This framework is a set of classes that implement the basic structure of a mesh computation, subject to the parameters for the Mesh pattern template. This structural framework defines the classes of the application and the flow of

control between the instances of these classes. The user does not add code directly to the framework, but rather creates subclasses of the framework classes to provide application–dependent implementations of "hook" methods. The framework provides the structure of the application and invokes these user–supplied hook methods. This is different than a library, where the user provides the structure of the application and a library provides utility routines. A framework provides design reuse by clearly separating the application–independent framework structure from the application–dependent code. The use of frameworks can reduce the effort required to build applications [2].

The Patterns Layer of CO_2P_3S emphasizes correctness. Generating the correct parallel structural code for a pattern template is only part of this effort. Once this structural code is created, CO_2P_3S also hides the structure of the framework so that it cannot be modified. This prevents users from introducing errors. Also, to ensure that users implement the correct hook methods, CO_2P_3S provides template viewers, shown in Figure 2(b), to display and edit these methods. At the Patterns Layer, the user can only implement these methods using the viewers. The user cannot modify the internals of the framework and introduce parallel structural errors. To further reduce the risk of programmer errors, the framework encapsulates all necessary synchronization for the provided parallel structure. The user does not need to include any synchronization or parallel code in the hook methods for the framework to operate correctly. The hook methods are normal, sequential code. These restrictions are relaxed in the lower layers of CO_2P_3S.

For the Mesh framework, the user must write two application–specific sections of code. The first part is the mainline method. A sample mainline is generated with the Mesh framework, but the user will likely need to modify the code to provide correct values for the application. The second part is the implementation of the mesh element class. The mesh element class defines the application–specific parts of the mesh computation: how to instantiate a mesh element, the mesh operation that the framework is parallelizing, the termination conditions for the computation, and a gather operation to collect the final results. The structural part of the Mesh framework creates a two–dimensional surface of these mesh elements and implements the flow of control through a parallel mesh computation. This structure uses the application–specific code supplied by the user for the specifics of the computation.

The mainline code is responsible for instantiating the mesh class and launching the computation. The mesh class is responsible for creating the surface of mesh elements, using the dimensions supplied by the user. The user can also supply an optional *initializer* object, providing additional state to the constructor for each mesh object. In this example, the initializer is a random number generator so that the morphogens can be initialized with random concentrations. Analogously, the user can also supply an optional *reducer* object to collect the final results of the computation, by applying this object to each mesh element after the mesh computation has finished. Once the mesh computation is complete, the results can be accessed through the reducer object. Finally, the user specifies the number of threads that should be used to perform the computation. The user specifies the number of horizontal and vertical blocks, decomposing the surface into smaller pieces. Each block is assigned to a different thread. This information is supplied at run-time so that the user can quickly experiment with different surface

```
import java.util.Random ;
public class MorphogenPair
{
    protected Morphogen morph1, morph2 ;

    public MorphogenPair(int x,int y,int width,int height,
                Object initializer) {
        Random gen = (Random) initializer;
        morph1 = new Morphogen(1.0-(gen.nextDouble()*2.0),2.0,1.0);
        morph2 = new Morphogen(1.0-(gen.nextDouble()*2.0),2.0,1.5);
    } /* MorphogenPair */

    public boolean notDone() {
        return(!(morph1.hasConverged() && morph2.hasConverged()));
    } /* notDone */

    public void prepare() {
        morph1.updateConcentrations();
        morph2.updateConcentrations();
    } /* prepare */

    public void interiorNode(MorphogenPair left, right, up, down) {
        morph1.simulate(left.getMorph1(),right.getMorph1(),
                up.getMorph1(),down.getMorph1(),morph2, 1);
        morph2.simulate(left.getMorph2(),right.getMorph2(),
                up.getMorph2(),down.getMorph2(),morph1, 2);
    } /* interiorNode */

    public void postProcess() {
        morph1.updateConcentrations();
    } /* postProcess */

    public void reduce(int x,int y,int width,int height,Object reducer) {
        Concentrations result = (Concentrations) reducer;
        result.concentration[x][y] = morph1.getConcentration() ;
    } /* reduce */
    // Define two accessors for the two morphogens.
} /* MorphogenPair */
```

Fig. 3. Selected parts of the `MorphogenPair` mesh element class.

sizes and numbers of threads. If these values were template parameters, the user would have to regenerate the framework and recompile the code for every new experiment.

The application–specific code in the Mesh framework is shown in Figure 3. The user writes an implementation of the mesh element class, `MorphogenPair`, defining methods for the specifics of the mesh computation for a single mesh element. When the framework is created, stubs for all of these methods are also generated. The framework iterates over the surface, invoking these methods for each mesh element at the appropriate time to execute the complete mesh computation. The sequence of method calls for the application is shown in Figure 4. This method is part of the structure of the Mesh framework, and is not available to the user at the Patterns Layer. However, this code shows the flow of control for a generic mesh computation. Using this code with the `MorphogenPair` implementation of Figure 3 shows the separation between the application–dependent and application–independent parts of the generated frameworks.

The constructor for the mesh element class (in Figure 3) creates a single element. The `x` and `y` arguments provide the location of the mesh element on the surface, which is of dimensions `width` by `height` (from the constructor for the mesh object). Pro-

```
public void meshMethod() {
    this.initialize();
    while(this.notDone()) {
        this.prepare();
        this.barrier();
        this.operate();
    } /* while */
    this.postProcess() ;
} /* meshMethod */
```

Fig. 4. The main loop for each thread in the Mesh framework.

viding these arguments allows the construction of the mesh element to take its position into account if necessary. The constructor also accepts the initializer object, which is applied to the new mesh element. In this example, the initializer is a random number generator used to create morphogens with random initial concentrations.

The `initialize()` method is used for any initialization of the mesh elements that can be performed in parallel. In the reaction–diffusion application, no such initialization is required, so this method does not appear in Figure 3.

The `notDone()` method must return true if the element requires additional iterations to complete its calculation and false if the computation has completed for the element. Typical mesh computations iterate until the values in the mesh elements converge to a final solution. This requires that a mesh element remember both its current value and the value from the previous iteration. The reaction–diffusion problem also involves convergence, so each morphogen has instance variables for both its current concentration and the previous concentration. When the difference between these two values falls below a threshold, the element returns false. By default, the stub generated for this method returns false, indicating that the mesh computation has finished.

The `interiorNode(left,right,up,down)` method performs the mesh computation for the current element based on its value and the values of the supplied neighbouring elements. This method is invoked indirectly from the `operate()` method of Figure 4. There are, in fact, up to nine different operations that could be performed by a mesh element, based on the location of the element on the surface and the boundary conditions. These different operations have a different set of available neighbouring elements. For instance, two other operations are `topLeftCorner(right,down)` and `rightEdge(left,up,down)`. Stubs are generated for every one of the nine possible operations that are required by the boundary conditions selected by the user in the Mesh pattern template. For the reaction–diffusion example, the boundary conditions are fully–toroidal, so every element is considered an interior node (as each element has all four available neighbours since the edges of the surface wrap around). This method computes the new values for the concentrations of both of its morphogen objects, based on its own concentration and that of its neighbours.

The `prepare()` method performs whatever operations are necessary to prepare for the actual mesh element computation just described. When an element computes its new values, it depends on state from neighbouring elements. However, these elements may be concurrently computing new values. In some mesh computations, it is important that the elements supply the value that was computed during the previous iteration of the

computation, not the current one. Therefore, each element must maintain two copies of its value, one that is updated during an iteration and another that is read by neighbouring elements. We call these states the write and read states. When an element requests the state from a neighbour, it gets the read state. When an element updates its state, it updates the write state. Before the next iteration, the element must update its read state with the value in its write state. The `prepare()` method can be used for this update. The reaction–diffusion example uses a read and write state for the concentrations in the two morphogen objects, which are also used in the `notDone()` method to determine if the morphogen has converged to its final value.

The `postProcess()` method is used for any postprocessing of the mesh elements that can be performed in parallel. For this problem, we use this method to update the read states before the final results are gathered, so that the collected results will be the concentrations computed in the last iteration of the computation.

The `reduce(x,y,width,height,reducer)` method applies a reducer object to the mesh elements to obtain the final results of the computation. This reducer is typically a container object to gather the results so that they can be used after the computation has finished. In this application, the reducer is an object that contains an array for the final concentrations, which is used to display the final texture. Like the initializer, the reducer is passed as an `Object` and must be cast before it can be used.

2.3 The Implementation of the Mesh Framework

The user of a Mesh framework does not have to know anything about any other classes or methods. However, in this section we briefly describe the structure of the Mesh framework. This is useful from both a scientific standpoint and for any advanced user who wants to modify the framework code by working at the Intermediate Code Layer.

In general, the granularity of a mesh computation at an individual element is too small to justify a separate thread or process for that element. Therefore, the two–dimensional surface of the mesh is decomposed into a rectangular collection of block objects (where the number of blocks is specified by a user in the mesh object constructor). Each block object is assigned to a different thread to perform the mesh computations for the elements in that block. We obtain parallelism by allowing each thread to concurrently perform its local computations, subject to necessary synchronization. The code executed by each thread, for its block, is `meshMethod()` from Figure 4.

We now look at the method calls in `meshMethod()`. The `initialize()`, `prepare()`, and `postProcess()` methods iterate over their block and invoke the method with the same name on each mesh element. The `notDone()` method iterates over each element in its block, calling `notDone()`. Each thread locally reduces the value returned by its local block to determine if the computation for the block is complete. If any element returns true, the local computation has not completed. If all elements return false, the computation has finished. The threads then exchange these values to determine if the whole mesh computation has finished. Only when all threads have finished does the computation end. The `barrier()` invokes a barrier, causing all threads to finish preparing for the iteration before computing the new values for their block. The user does not implement any method for a mesh element corresponding to this method. Chaotic meshes do not include this synchronization. The `operate()`

Table 1. Speedups and wall clock times for the reaction–diffusion example.

	Processors	2	4	8	16
1680 by 1680	Speedup	1.75	3.13	4.92	6.50
	Time (sec)	5734	3008	1910	1448

method iterates over the mesh elements in the block, invoking the mesh operation method for that single element with the proper neighbouring elements as arguments. However, since some of the elements are interior elements and some are on the boundary of the mesh, there are up to nine different methods that could be invoked. The most common method is `interiorNode(left,right,up,down)`, but other methods may exist and may also be used, depending on the selected boundary conditions. The method is determined using a Strategy pattern [1] that is generated with the framework. Note that elements on the boundary of a block have neighbours that are in other blocks so they will invoke methods on elements in other blocks.

2.4 Evaluating the Mesh Framework

The performance of the reaction–diffusion example is shown in Table 1. These performance numbers are not necessarily the best that can be obtained. They are meant to show that for a little effort, it is possible to write a parallel program and quickly obtain speedups. Once we decide to use the Mesh pattern template, the structural code for the program is generated in a matter of minutes. Using existing sequential code, the remainder of the application can be implemented in several hours. To illustrate the relative effort required, we note that of the 696 lines of code for the parallel program, the user was responsible for 212 lines, about 30%. Of the 212 lines of user code, 158 lines, about 75%, was reused from the sequential version. We must point out, though, that these numbers are a function of the problem being solved, and not a function of the programming system. However, the generated code is a considerable portion of the overall total for the parallel program. Generating this code automatically reduces the effort needed to write parallel programs.

The program was run using a native threaded Java interpreter from SGI with optimizations and JIT turned on. The execution environment was an SGI Origin 2000 with 195MHz R10000 processors and 10GB of RAM. The virtual machine was started with 512MB of heap space. The speedups are based on wall clock times compared to a sequential implementation. These speedup numbers only include computation time.

From the table, we can see that the problem scales well up to four processors, but the speedup drops off considerably thereafter. The problem is granularity; as more processors are added, the amount of computation between barrier points decreases until synchronization is a limiting factor in performance. Larger computations, with either a larger surface or a more complex computation, yield better speedups.

3 Other Patterns in CO_2P_3S

In addition to the Mesh, CO_2P_3S supports several other pattern templates. Two of these are the Phases and the Distributor. The Phases template provides an extendible way

to create phased algorithms. Each phase can be parallelized individually, allowing the parallelism to change as the algorithm progresses. The Distributor template supports a data–parallel style of computation. Methods are invoked on a parent object, which forwards the same method to a fixed number of child objects, each executing in parallel.

We have composed these two patterns to implement the parallel sorting by regular sampling algorithm (PSRS) [6]. The details on the implementation of this program are in [4]. To summarize the results, the complete program was 1507 lines of code, with 669 lines (44%) written by the user. 212 of the 669 lines is taken from the JGL library. There was little code reuse from the sequential version of the problem as PSRS is an explicitly parallel algorithm. Because this algorithm does much less synchronization, it scales well up to 16 processors, obtaining a speedup of 11.2 on 16 processors.

4 Conclusions

This paper presented the graphical user interface of the CO_2P_3S parallel programming system. In particular, it showed the development of a reaction–diffusion texture generation program using the Mesh parallel design pattern template, using the facilities provided at the highest layer of abstraction in CO_2P_3S. Our experience suggests that we can quickly create a correct parallel structure that can be used to write a parallel program and obtain performance benefits.

Acknowledgements

This research was supported by grants and resources from the Natural Science and Engineering Research Council of Canada, MACI (Multimedia Advanced Computational Infrastructure), and the Alberta Research Council.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.
2. R. Johnson. Frameworks = (components + patterns). *CACM*, 40(10):39–42, October 1997.
3. S. MacDonald, J. Schaeffer, and D. Szafron. Pattern–based object–oriented parallel programming. In *Proceedings of ISCOPE'97*, LNCS volume 1343, pages 267–274, 1997.
4. S. MacDonald, D. Szafron, and J. Schaeffer. Object–oriented pattern–based parallel programming with automatically generated frameworks. In *Proceedings of COOTS'99*, pages 29–43, 1999.
5. B. Massingill, T. Mattson, and B. Sanders. A pattern language for parallel application programming. Technical Report CISE TR 99–009, University of Florida, 1999.
6. H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.
7. A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.
8. A. Witkin and M. Kass. Reaction–diffusion textures. *Computer Graphics (SIGGRAPH '91 Proccedings)*, 25(4):299–308, July 1991.

Topic 02
Performance Evaluation and Prediction

Thomas Fahringer and Wolfgang E. Nagel

Topic Chairmen

Even today, when microprocessor vendors announce breakthroughs every other week, performance evaluation is still one of the key issues in parallel computing. One of the observations is that on a single PE, many, if not most applications relevant to the technical field do not benefit adequately from clock rate improvement. The reason for this is memory access: most data read and write operations access memory which is relatively slow compared to processor speed. With several levels of caches we now have complex system architectures which, in principal, provide plenty of options to keep the data as close as possible to the processor. Nevertheless, compiler development proceeds in slow progression, and the single PE still dominates the results achieved on large parallel machines.

In a couple of months, almost every vendor will offer very powerful SMP nodes with impressive system peak performance numbers, based on multiplication of single PE peak performance numbers. These SMP nodes will be coupled to larger systems with even more impressive peak numbers. In contrast, the sustained performance for real applications is far from satisfactory, and a large number of application programmers are struggling with many complex features of modern computer architectures.

This topic aims at bringing together people working in the different fields of performance modeling, evaluation, prediction, measurement, benchmarking, and visualization for parallel and distributed applications and architectures. It covers aspects of techniques, implementations, tools, standardization efforts, and characterization and performance-oriented development of distributed and parallel applications. Especially welcome have been contributions devoted to performance evaluation and prediction of object-oriented and/or multi-threaded programs, novel and heterogeneous architectures, as well as web-based systems and applications. 27 papers were submitted to this workshop, 8 were selected as regular papers and 6 as short papers.

Performance Diagnosis Tools

More intelligent tools which sometimes even work automatically will have a strong impact on the acceptance and use of parallel computers in future. This session is devoted to that research field. The first paper introduces a new technique for automated performance diagnosis using the program's call graph. The implementation is based on a new search strategy and a new dynamic instrumentation to resolve pointer-based dynamic call sites at run-time. The second

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 105–107, 2000.
© Springer-Verlag Berlin Heidelberg 2000

paper presents a class library for detecting typical performance problems in event traces of MPI applications. The library is implemented using the powerful high-level trace analysis language EARL and is embedded in the extensible tool component called EXPERT. The third contribution in this session describes Pajé, an interactive visualization tool for displaying the execution of parallel applications where a (potentially) large number of communicating threads of various life-times execute on each node of a distributed memory parallel system.

Performance Prediction and Analysis

The efficient usage of resources – either memory or just processors – should be a prerequisite for all kinds of parallel programming. If multiple users have different requests over time, the scheduling and allocation of resources to jobs becomes a critical issue. This session summarizes contributions in that field. The first paper presents a hybrid version of two previous models that perform analysis of memory hierarchies. It combines the positive features of both models by interleaving the analysis methods. Furthermore, it links the models to provide a more focused method for analyzing performance contributions due to latency hiding techniques such as outstanding misses and speculative execution. The second paper describes the tool set PACE that provides detailed predictive performance information throughout the implementation and execution stages of an application. Because of the relatively fast analysis times, the techniques presented can also be used at run-time to assist in application steering and the efficient management of the available system resources. The last paper addresses the problem of estimating the total execution time of a parallel program-based domain decomposition.

Performance Prediction and Simulation

The third part of the workshop covers a couple of important aspects ranging from performance prediction to cache simulation. The first paper in this session describes a technique for deriving performance models from design patterns expressed in the Unified Modeling Language (UML) notation. The second paper describes the use of an automatic performance analysis tool for describing the behaviour of a parallel application. The third paper proposes a new cost-effective approach for on-the-fly micro-architecture simulations using long running applications. The fourth paper introduces a methodology for a comprehensive examination of workstation cluster performance and proposes a tailored benchmark evaluation tool for clusters. The fifth paper investigates performance prediction for a discrete-event simulation program. The performance analyzer tries to predict what the speedups will be, if a conservative, “super-step” (synchronous) simulation protocol is used. The last paper in this session focuses on cache memories, cache miss equations, and sampling.

Performance Modeling of Distributed Systems

Performance analysis and optimization is even more difficult if the environment is physically distributed and possibly heterogeneous. The first paper studies the influence of process mapping on message passing performance on Cray T3E and the Origin 2000. First, the authors have designed an experiment where processes are paired off in a random manner and messages of different sizes are exchanged between them. Thereafter, they have developed a mapping algorithm for the T3E, suited to n-dimensional cartesian topologies. The second paper focuses on heterogeneity in Networks of Workstations (NoW). The authors have developed a performance prediction tool called ChronosMix, which can predict the execution time of a distributed algorithm on parallel or distributed architecture.

A Callgraph-Based Search Strategy for Automated Performance Diagnosis[1]

Harold W. Cain Barton P. Miller Brian J.N. Wylie
{cain,bart,wylie}@cs.wisc.edu
http://www.cs.wisc.edu/paradyn

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685, U.S.A.

Abstract. We introduce a new technique for automated performance diagnosis, using the program's callgraph. We discuss our implementation of this diagnosis technique in the Paradyn Performance Consultant. Our implementation includes the new search strategy and new dynamic instrumentation to resolve pointer-based dynamic call sites at run-time. We compare the effectiveness of our new technique to the previous version of the Performance Consultant for several sequential and parallel applications. Our results show that the new search method performs its search while inserting dramatically less instrumentation into the application, resulting in reduced application perturbation and consequently a higher degree of diagnosis accuracy.

1 Introduction

Automating any part of the performance tuning cycle is a valuable activity, especially where intrinsically complex and non-deterministic distributed programs are concerned. Our previous research has developed techniques to automate the location of performance bottlenecks [4,9], and other tools can even make suggestions as to how to fix the program to improve its performance [3,8,10]. The Performance Consultant (PC) in the Paradyn Parallel Performance Tools has been used for several years to help automate the location of bottlenecks. The basic interface is a one-button approach to performance instrumentation and diagnosis. Novice programmers immediately get useful results that help them identify performance-critical activities in their program. Watching the Performance Consultant in operation also acts as a simple tutorial in strategies for locating bottlenecks. Expert programmers use the Performance Consultant as a head start in diagnosis. While it may not find some of the more obscure problems, it saves the programmer time in locating the many common ones.

An important attribute of the Performance Consultant is that it uses dynamic instrumentation [5,11] to only instrument the part of the program in which it is currently in-

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B504964, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 108-122, 2000.
© Springer-Verlag Berlin Heidelberg 2000

terested. When instrumentation is no longer needed, it is removed. Insertion and removal of instrumentation occur while the program is running unmodified executables. Instrumentation can include simple counts (such as function calls or bytes of I/O or communication), process and elapsed times, and blocking times (I/O and synchronization).

While the Performance Consultant has shown itself to be useful in practice, there are several limitations that can reduce its effectiveness when operating on complex application programs (with many functions and modules). These limitations manifest themselves when the PC is trying to isolate a bottleneck to a particular function in an application's code. The original PC organized code into a static hierarchy of modules and functions within modules. An automated search based on such a tree is a poor way to direct a search for bottlenecks, for several reasons: (1) when there is a large number of modules, it is difficult to know which ones to examine first, (2) instrumenting modules is expensive, and (3) once a bottleneck is isolated to a module, if there is a large number of functions within a module, it is difficult to know which ones to examine first.

In this paper, we describe how to avoid these limitations by basing the search on the application's callgraph. The contributions of this paper include an automated performance diagnostic strategy based on the application's callgraph, new instrumentation techniques to discover callgraph edges in the presence of function pointers, and a demonstration of the effectiveness of these new techniques. Along with the callgraph-based search, we are able to use a less expensive form of timing primitive, reducing run-time overhead.

The original PC was designed to automate the steps that an experienced programmer would naturally perform when trying to locate performance-critical parts of an application program. Our callgraph enhancements to the PC further this theme. The general idea is that isolating a problem to a part of the code starts with consideration of the main program function and if it is found to be critical, consideration passes to each of the functions it calls directly; for any of those found critical, investigation continues with the functions that they in turn call. Along with the consideration of called functions, the caller must also be further assessed to determine whether or not it is a bottleneck in isolation. This repeats down each exigent branch of the callgraph until all of the critical functions are found.

The callgraph-directed Performance Consultant is now the default for the Paradyn Parallel Performance tools (as of release 3.0). Our experience with this new PC has been uniformly positive; it is both faster and generates significantly less instrumentation overhead. As a result, applications that previously were not suitable for automated diagnosis can now be effectively diagnosed.

The next section describes Paradyn's Performance Consultant in its original form, and later Section 4 describes the new search strategy based on the application program's callgraph. The callgraph-based search needs to be able to instrument and resolve function pointers, and this mechanism is presented in Section 3. We have compared the effectiveness of the new callgraph-based PC to the original version on several serial and parallel applications, and the experiments and results are described in Section 5.

2 Some Paradyn Basics

Paradyn [9] is an application profiler that uses *dynamic instrumentation* [5,6,11] to insert and delete measurement instrumentation as a program runs. Run-time selection of instrumentation results in a relatively small amount of data compared to static (compile or link time) selection. In this section, we review some basics of Paradyn instrumentation, then discuss the Performance Consultant and its original limitations when trying to isolate a bottleneck to particular parts of a program's code.

2.1 Exclusive vs. Inclusive Timing Metrics

Paradyn supports two types of timing metrics, *exclusive* and *inclusive*. Exclusive metrics measure the performance of functions in isolation. For example, exclusive CPU time for function `foo` is only the time spent in that function itself, excluding its callees. Inclusive metrics measure the behavior of a function while it is active on the stack. For example, inclusive time for a function `foo` is the time spent in `foo`, including its callees. Timing metrics can measure process or elapsed (wall) time, and can be based on CPU time or I/O, synchronization, or memory blocking time.

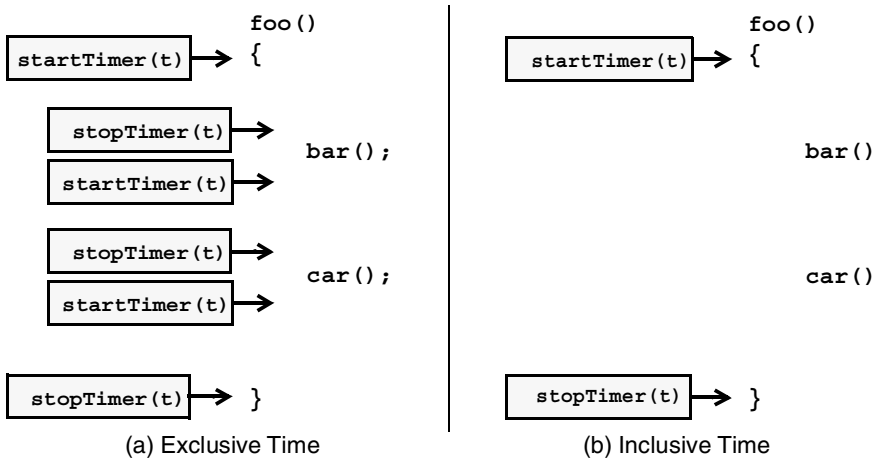


Figure 1 Timing instrumentation for function `foo`.

Paradyn inserts instrumentation into the application to make these measurements. For exclusive time (see Figure 1a), instrumentation is inserted to start the timer at the function's entry and stop it at the exit(s). To include only the time spent in this function, we also stop the timer before each function call site and restart it after returning from the call. Instrumentation for inclusive time is simpler; we only need to start and stop the timer at function entry and exit (see Figure 1b). This simpler instrumentation also generates less run-time overhead. A start/stop pair of timer calls takes 56.5 µs on a SGI Origin. The savings become more significant in functions that contain many call sites.

2.2 The Performance Consultant

Paradyn's Performance Consultant (PC) [4,7] dynamically instruments a program with timer start and stop primitives to automate bottleneck detection during program execu-

tion. The PC starts searching for bottlenecks by issuing instrumentation requests to collect data for a set of pre-defined performance *hypotheses* for the whole program. Each hypothesis is based on a continuously measured value computed by one or more Paradyn metrics, and a fixed threshold. For example, the PC starts its search by measuring total time spent in computation, synchronization, and I/O waiting, and compares these values to predefined thresholds. Instances where the measured value for the hypothesis exceeds the threshold are defined as *bottlenecks*. The full collection of hypotheses is organized as a tree, where hypotheses lower in the tree identify more specific problems than those higher up.

We represent a program as a collection of discrete program resources. Resources include the program code (e.g., modules and functions), machine nodes, application processes and threads, synchronization objects, data structures, and data files. Each group of resources provides a distinct view of the application. We organize the program resources into trees called *resource hierarchies*, the root node of each hierarchy labeled with the hierarchy's name. As we move down from the root node, each level of the hierarchy represents a finer-grained description of the program. A *resource name* is formed by concatenating the labels along the unique path within the resource hierarchy from the root to the node representing the resource. For example, the resource name that represents function `verifyA` (shaded) in Figure 2 is < /Code/testutil.C/verifyA>.

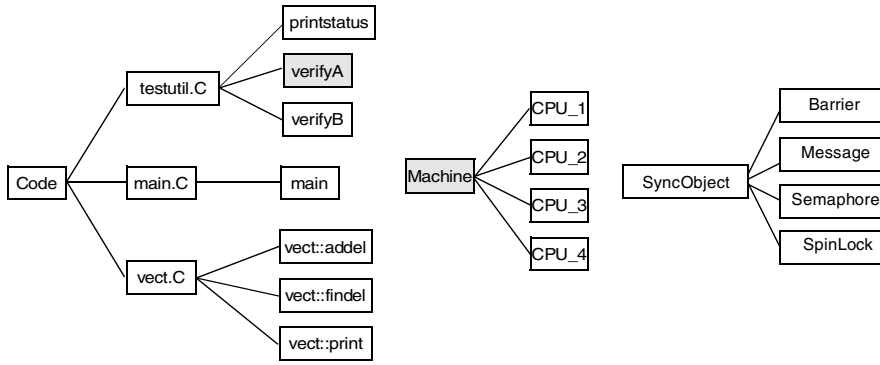


Figure 2 Three Sample Resource Hierarchies: Code, Machine, and SyncObject.

For a particular performance measurement, we may wish to isolate the measurement to specific parts of a program. For example, we may be interested in measuring I/O blocking time as the total for one entire execution, or as the total for a single function. A *focus* constrains our view of the program to a selected part. Selecting the root node of a resource hierarchy represents the unconstrained view, the whole program. Selecting any other node narrows the view to include only those leaf nodes that are immediate descendents of the selected node. For example, the shaded nodes in Figur e2 represent the constraint: code function `verifyA` running on any CPU in the machine, which is labeled with the focus: < /Code/testutil.C/verifyA, /Machine >.

Each node in a PC search represents instrumentation and data collection for a (hypothesis : focus) pair. If a node tests true, meaning a bottleneck has been found, the

Performance Consultant tries to determine more specific information about the bottleneck. It considers two types of search expansion: a more specific hypothesis and a more specific focus. A child focus is defined as any focus obtained by moving down along a single edge in one of the resource hierarchies. Determining the children of a focus by this method is referred to as *refinement*. If a pair (h : f) tests false, testing stops and the node is not refined. The PC refines all true nodes to as specific a focus as possible, and only these foci are used as roots for refinement to more specific hypothesis constructions (to avoid undesirable exponential search expansion).

Each (hypothesis : focus) pair is represented as a node of a directed acyclic graph called the Search History Graph (SHG). The root node of the SHG represents the pair (TopLevelHypothesis : WholeProgram), and its child nodes represent the refinements chosen as described above. An example Paradyn SHG display is shown in Figur e3.

2.3 Original Paradyn: Searching the Code Hierarchy

The search strategy originally used by the Performance Consultant was based on the module/function structure of the application. When the PC wanted to refine a bottleneck to a particular part of the application code, it first tried to isolate the bottleneck to particular modules (`.o/.so/.obj/.dll` file). If the metric value for the module is above the threshold, then the PC tries to isolate the bottleneck to particular functions within the module.

This strategy has several drawbacks for large programs.

1. Programs often have many modules; and modules often have hundreds of functions. When the PC starts to instrument modules, it cannot instrument all of them efficiently at the same time and has no information on how to choose which ones to instrument first; the order of instrumentation essentially becomes random. As a result, many functions are needlessly instrumented. Many of the functions in each module may never be called, and therefore do not need to be instrumented. By using the callgraph, the new PC operates well for any size of program.
2. To isolate a bottleneck to a particular module or function, the PC uses exclusive metrics. As mentioned in Section 2.1, these metrics require extra instrumentation code at each call site in the instrumented functions. The new PC is able to use the cheaper inclusive metrics.
3. The original PC search strategy was based on the notion that coarse-grain instrumentation was less expensive than fine-grained instrumentation. For code hierarchy searches, this means that instrumentation to determine the total time in a module should be cheaper than determining the time in each individual function. Unfortunately, the cost of instrumenting a module is the same as instrumenting all the functions in that module. The only difference is that we have one timer for the entire module instead of one for each function.

   This effect could be reduced for module instrumentation by not stopping and starting timers at call sites that call functions inside the same module. While this technique is possible, it provides such a small benefit, it was not worth the complexity. Use of the callgraph in the new PC avoids using the code hierarchy.

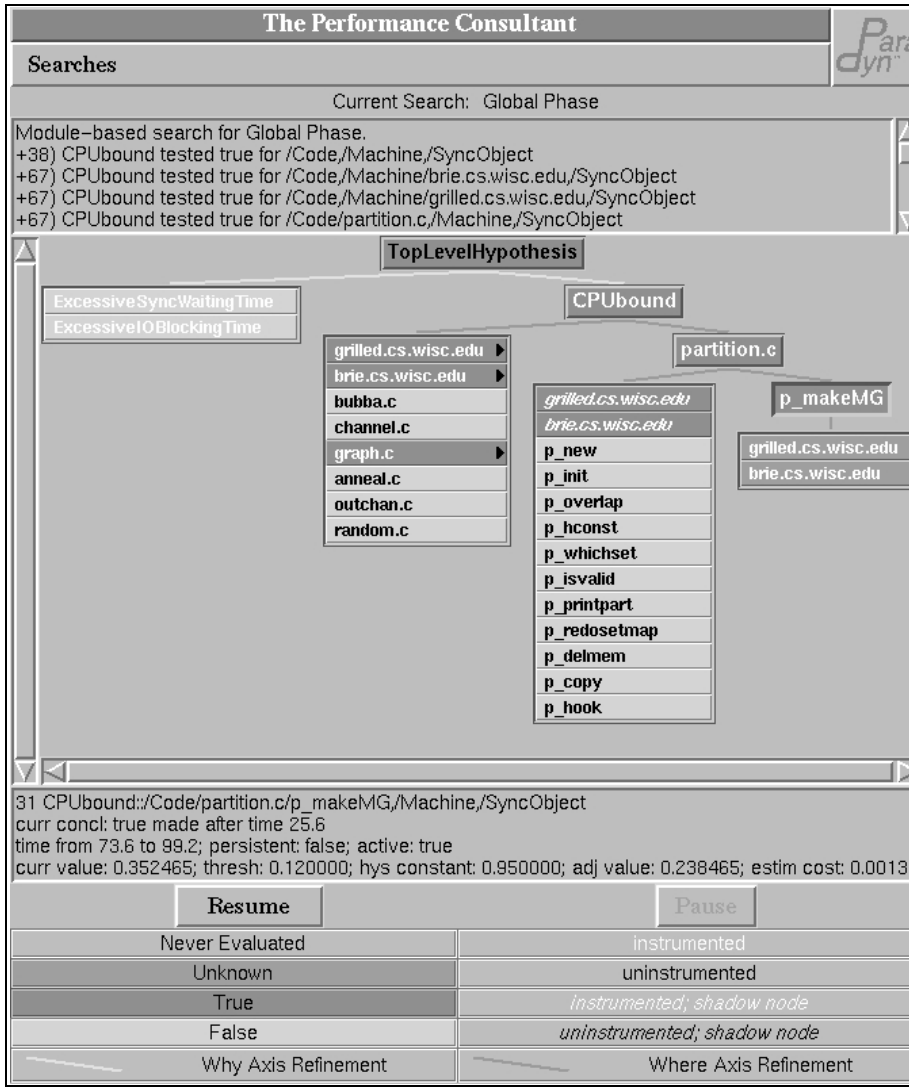


Figure 3 The Original Performance Consultant with a Bottleneck Search in Progress. The three items immediately below **TopLevelHypothesis** have been added as a result of refining the hypothesis. **ExcessiveSyncWaitingTime** and **ExcessiveIOBlocking-Time** have tested false, as indicated by node color (pink or light grey), and **CPUbound** (blue or dark grey) has tested true and been expanded by refinement. Code hierarchy module nodes **bubba.c**, **channel.c**, **anneal.c**, **outchan.c**, and **random.c** all tested false, whereas modules **graph.c** and **partition.c** and Machine nodes **grilled** and **brie** tested true and were refined. Only function **p_makeMG** in module **partition.c** was subsequently found to have surpassed the bottleneck hypothesis threshold, and the final stage of the search is considering whether this function is exigent on each Machine node individually. (Already evaluated nodes with names rendered in black no longer contain active instrumentation, while instrumented white-text nodes continue to be evaluated.)

3 Dynamic Function Call Instrumentation

Our search strategy is dependent on the completeness of the callgraph used to direct the search. If all caller-callee relationships are not included in this graph, then our search strategy will suffer from blind spots where this information is missing. Paradyn's standard start-up operation includes parsing the executable file in memory (dynamically linked libraries are parsed as they are loaded). Parsing the executable requires identifying the entry and exits of each function (which is trickier than it would appear [6]) and the location of each function call site. We classify call sites as *static* or *dynamic*. Static sites are those whose destination we can determine from inspection of the code. Dynamic call sites are those whose destination is calculated at run-time. While most call sites are static, there is still a non-trivial number of dynamic sites. Common sources of dynamic call sites are function pointers and C++ virtual functions.

Our new instrumentation resolves the address of the callee at dynamic call sites by inserting instrumentation at these sites. This instrumentation computes the appropriate callee address from the register contents and the offsets specified in the call instruction. New call destination addresses are reported to the Paradyn front-end, which then updates its callgraph and notifies the PC. When the PC learns of a new callee, it incorporates the callee in its search. We first discuss the instrumentation of the call site, then discuss how the information gathered from the call site is used.

3.1 Call Site Instrumentation Code

The Paradyn daemon includes a code generator to dynamically generate machine-specific instrumentation code. As illustrated in Figure 4, instrumentation code is inserted

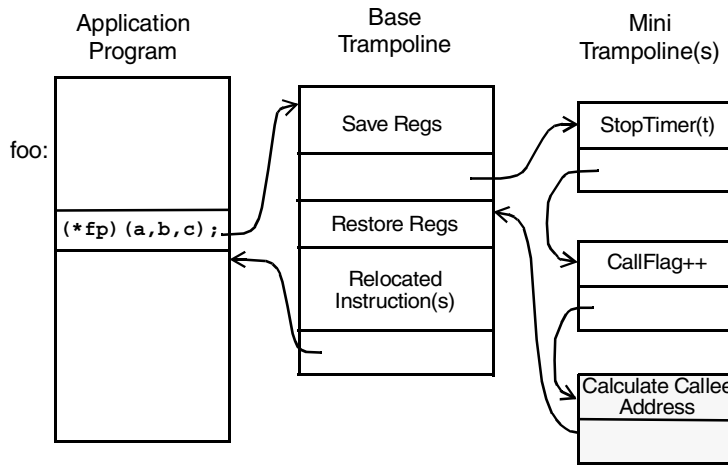


Figure 4 Simplified Control Flow from Application to Instrumentation Code. A dynamic call instruction in function `foo` is replaced with branch instructions to the base trampoline. The base trampoline saves the application's registers and branches to a series of mini trampolines that each contain different instrumentation primitives. The final mini trampoline returns to the base trampoline, which restores the application's registers, emulates the relocated dynamic call instruction, and returns control to the application.

into the application by replacing an instruction with a branch to a code snippet called the *base-trampoline*. The base-trampoline saves and restores the application's state before and after executing instrumentation code. The instrumentation code for a specific primitive (e.g. a timing primitive) is contained in a *mini-trampoline*.

Dynamic call instructions are characterized by the destination address residing in a register or (sometimes on the x86) a memory location. The dynamic call resolution instrumentation code duplicates the address calculation of these call instructions. This code usually reads the contents of a register. This reading is slightly complicated, since (because we are instrumenting a call instruction) there are two levels of registers saved: the caller-saved registers as well as those saved by the base trampoline. The original contents of the register have been stored on the stack and may have been overwritten. To access these saved registers, we added a new code generator primitive (abstract syntax tree operand type). We have currently implemented dynamic call site determination for the MIPS, SPARC, and x86, with Power2 and Alpha forthcoming.

A few examples of the address calculations are shown in Table 1. We show an example of the type of instruction that would be used at a dynamic call site, and mini-trampoline code that would retrieve the saved register or memory value and calculate the callee's address.

Table 1: Dynamic callee address calculation.

Instruction Set	Call Instruction	Mini-Trampoline Address Calculation	Explanation
MIPS	`jalr $t9`	`ld $t0, 48($sp)`	Load `$t9` from stack.
x86	`call [%edi]`	`mov %eax,-160(%ebp)` `mov %ecx,[%eax]`	Load `%edi` from stack. Load function address from memory location pointed to by `%eax`.
SPARC	`jmpl %l0,%o7`	`ld [%fp+20],%l0` `add %l0,%i7,%l3`	Load `%l0` from stack. `%o7` becomes `%i7` in new register window.

3.2 Control Flow for Dynamic Call Site Instrumentation

To instrument a dynamic call site, the application is paused, instrumentation inserted, and the application resumed. The call site instrumentation is inserted on demand, i.e., only when the caller function becomes relevant to the bottleneck search. For example, if function `foo` contains a dynamic call site, this site is not instrumented until the PC identifies `foo` as a bottleneck, at which point we need to know all of the callees of `foo`. By instrumenting these sites on demand, we minimize the amount of instrumentation in the application. The flow of control for these steps is shown as steps 1 and 2 in Figure 5.

The call site instrumentation detects the first time that a callee is called from that site. When a callee is first called from a site, the instrumentation code notifies the Paradyn daemon (step A in Figur e5), which notifies the PC in the Paradyn front-end (step

B). The new caller-callee edge is added to the callgraph and, if desired, instrumentation can be inserted in the newly discovered callee (steps C and D).

We do not want to incur this communication and instrumentation cost each time that a dynamic call site is executed. Fortunately, most call sites only call a few different functions, so we keep a small table of callee addresses for each dynamic call site. Each time that a dynamic call site is executed and a callee determined, we check this table to see if it is a previously known caller-callee pair. If the pair has been previously seen, we bypass the steps A-D.

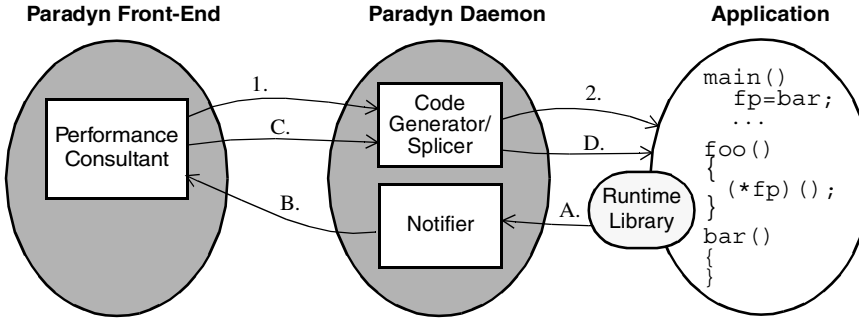


Figure 5 Control Flow between Performance Consultant and Application. (1) PC issues dynamic call-site instrumentation request for function `foo`. (2) Daemon instruments dynamic call sites in `foo`. (A) Application executes call instruction and when a new callee is found, runtime library notifies daemon. (B) Daemon notifies PC of new callee `bar` for function `foo`. (C) PC requests inclusive timing metric for function `bar`. (D) Daemon inserts timing instrumentation for `bar`.

Once the Paradyn daemon has notified the Performance Consultant of a new dynamic caller-callee relationship, the PC can take advantage of this information. If the dynamic caller has been previously determined a bottleneck, then the callee must be instrumented to determine if it is also a bottleneck. A possible optimization to this sequence is for the Paradyn daemon to instrument the dynamic callee as soon as it is discovered, thus reducing the added delay of conveying this information to the Paradyn front-end and waiting for the front-end to issue an instrumentation request for the callee. However, this optimization would require knowledge by the Paradyn daemon of the type of instrumentation timing primitive desired for this callee, and would also limit the generality of our technique for dynamic callee determination.

4 Callgraph-Based Searching

We have modified the Performance Consultant's code hierarchy search strategy to direct its search using the application's callgraph. The remainder of the search hierarchies (such as Machine and SyncObject) are still searched using the structure of their hierarchy; the new technique is only used when isolating the search to a part of the Code hierarchy.

When the PC starts refining a potential bottleneck to a specific part of the code, it starts at the top of the program graph, at the program entry function for each distinct

executable involved in the computation. These functions are instrumented to collect an inclusive metric. For example, if the current candidate bottleneck were CPUbound, the function would be instrumented with the CPU time inclusive metric. The timer associated with this metric runs whenever the program is running and this function was on the stack (presumably, the entire time that the application is running in this case).

If the metric value for the main program function is above the threshold, the PC uses the callgraph to identify all the functions called from it, and each is similarly instrumented with the same inclusive metric. In the callgraph-based search, if a function's sustained metric value is found to be below the threshold, we stop the search for that branch of the callgraph (i.e., we do not expand the search to include the function's children). If the function's sustained metric value is above the threshold, the search continues by instrumenting the functions that it calls. The search in the callgraph continues in this manner until all possible branches have been exhausted either because the accumulated metric value was too small or we reached the leaves of the callgraph.

Activating instrumentation for functions currently executing, and therefore found on the callstack, requires careful handling to ensure that the program and instrumentation (e.g., active timers and flags) remain in a consistent state. Special retroactive instrumentation needs to be immediately executed to set the instrumentation context for any partially-executed and now instrumented function, prior to continuing program execution. Timers are started immediately for already executing functions, and consequently produce measurements earlier than waiting for the function to exit (and be removed from the callstack) before instrumenting it normally.

The callgraph forms a natural organizational structure for three reasons. First, a search strategy based on the callgraph better represents the process that an experienced programmer might use to find bottlenecks in a program. The callgraph describes the overall control flow of the program, following a path that is intuitive to the programmer. We do not instrument a function unless it is a reasonable candidate to be a bottleneck: its calling functions are currently be considered a bottleneck. Second, using the callgraph scales well to large programs. At each step of the search, we are addressing individual functions (and function sizes are typically not proportional to the overall code size). The total number of modules and functions do not effect the strategy. Third, the callgraph-based search naturally uses inclusive time metrics, which are (often significantly) less costly in a dynamic instrumentation system than their exclusive time counterparts.

An example of the callgraph-based Paradyn SHG display at the end of a comprehensive bottleneck search is shown in Figure 6.

While there are many advantages to using this callgraph-based search method, it has a few disadvantages. One drawback is that this search method has the potential to miss a bottleneck when a single resource-intensive function is called by numerous parent functions, yet none of its parents meet the threshold to be considered a bottleneck. For example, an application may spend 80% of its time executing a single function, but if that function has many parents, none of which are above the bottleneck threshold, our search strategy will fail to find the bottleneck function. To handle this situation, it is worth considering that the exigent functions are more than likely to be found on the

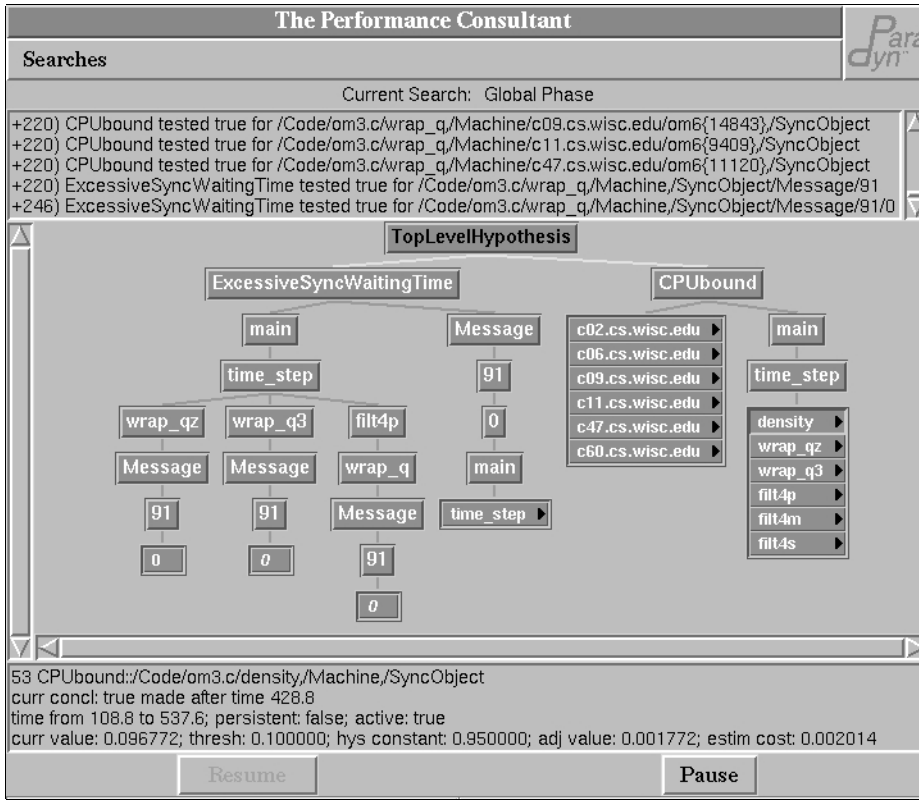


Figure 6 The Callgraph-based Performance Consultant after Search Completion. This snapshot shows the Performance Consultant upon completion of a search with the OM3 application when run on 6 Pentium II Xeon nodes of a Linux cluster. For clarity, all hypothesis nodes which tested false are hidden, leaving only paths which led to the discovery of a bottleneck. This view of the search graph illustrates the path that the Performance Consultant followed through the callgraph to locate the bottleneck functions. Six functions, all called from the **time_step** routine, have been found to be above the specified threshold to be considered CPU bottlenecks, both in aggregation and on each of the 6 cluster nodes. The **wrap_q**, **wrap_qz** and **wrap_q3** functions have also been determined to be synchronization bottlenecks when using MPI communicator **91** and message tag **0**.

stack whenever Paradyn is activating or modifying instrumentation (or if it were to periodically 'sample' the state of the callstack). A record kept of these ''callstack samples'' therefore forms an appropriate basis of candidate functions for explicit consideration, if not previously encountered, during or on completion of the callgraph-based search.

5 Experimental Results

We performed several experiments to evaluate the effectiveness of our new search method relative to the original version of the Performance Consultant. We use three criteria for our evaluation: the accuracy, speed, and efficiency with which the PC performs its search. The accuracy of the search is determined by comparing those bottlenecks reported by the Performance Consultant to the set of bottlenecks considered true application bottlenecks. The speed of a search is measured by the amount of time required for each PC to perform its search. The efficiency of a search is measured by the amount of instrumentation used to conduct a bottleneck search; we favor a search strategy that inserts less instrumentation into the application. We describe our experimental set-up and then present results from our experiments.

5.1 Experimental Setup

We used three sequential applications, a multithreaded application and two parallel application for these experiments. The sequential applications include the SPEC95 benchmarks fpppp (a Fortran application that performs multi-electron derivatives) and go (a C program that plays the game of Go against itself), as well as Draco (a Fortran hydrodynamic simulation of inertial confinement fusion, written by the laser fusion groups at the University of Rochester and University of Wisconsin). The sequential applications were run on a dual-processor SGI Origin under IRIX 6.5. The matrix application is based on the Solaris threads package and was run on an UltraSPARC Solaris 2.6 uniprocessor. The parallel application ssTwod solves the 2-D Poisson problem using MPI on four nodes of an IBM SP/2 (this is the same application used in a previous PC study[7]). The parallel application OM3 is a free-surface, *z*-coordinate general circulation ocean model, written using MPI by members of the Space Science and Engineering Center at the University of Wisconsin. OM3 was run on eight nodes of a 24-node SGI Origin under IRIX 6.5. Some characteristics of these applications that affect the Performance Consultant search space are detailed in Table 2. (All system libraries are explicitly excluded from this accounting and the subsequent searches.)

Table 2: Application search space characteristics.

Application (Language)	Lines of code	Number of modules	Number of functions	Number of dynamic call sites
Draco (F90)	61,788	232	256	5
go (C)	26,139	18	376	1
fpppp (F77)	2,784	39	39	0
matrix (C/Sthreads)	194	1	5	0
ssTwod (F77/MPI)	767	7	9	0
OM3 (C/MPI)	2,673	1	28	3

We ran each application program under two conditions: first, with the original PC, and then with the new callgraph-based PC. In each case, we timed the run from the start of the search until the PC had no more alternatives to evaluate. For each run, we saved the complete history of the performance search (using the Paradyn export facility) and recorded the time at which the PC found each bottleneck.

A 5% threshold was used for CPU bottlenecks and 12% threshold for synchronization waiting time bottlenecks. For the sequential applications, we verified the set of application bottlenecks using the prof profiling tool. For the parallel applications, we used Paradyn manual profiling along with both versions of the Performance Consultant to determine their bottlenecks.

5.2 Results

We ran both the original and modified versions of the Performance Consultant for each of the applications, measuring the time required to locate all of the bottlenecks. Each experiment is a single execution and search. For OM3, the SGI Origin was not dedicated to our use, but also not heavily loaded during the experiments. In some cases, the original version of the PC was unable to locate all of an application's bottlenecks due to the perturbation caused by the larger amount of instrumentation it requires. Tabl e3 shows the number of bottlenecks found by each version of the PC, and the time required to complete each search. As we can see, the size of an application has a significant impact on the performance of the original PC. For the small fpppp benchmark and matrix application, the original version of the PC locates the application's bottlenecks a little faster than the callgraph-based PC. This is because they have few functions and no complex bottlenecks (being completely CPUbound programs, there are few types and combinations of bottlenecks). As a result, the original Performance Consultant can quickly instrument the entire application. The new Performance Consultant, however, always has to traverse some portion of the application's callgraph.

Table 3: Accuracy, overhead and speed of each search method.

	Bottlenecks found in complete search		Instrumentation mini-tramps used		Required search time (seconds)	
Application	**Original**	**Callgraph**	**Original**	**Callgraph**	**Original**	**Callgraph**
Draco	3	5	14,317	228	1,006	322
go	2	4	12,570	284	755	278
fpppp	3	3	474	96	141	186
matrix	4	5	439	43	200	226
ssTwod	9	9	43,230	11,496	461	326
OM3	13	16	184,382	60,670	2,515	957

For the larger applications, the new search strategy's advantages are apparent. The callgraph-based Performance Consultant performs its search significantly faster for

each program other than fpppp and matrix. For Draco, go, and OM3, the original Performance Consultant's search not only requires more time, but due to the additional perturbation that it causes, it is unable to resolve some of the bottlenecks. It identifies only three of Draco's five bottleneck functions, two of go's four bottlenecks, and 13 of OM3's 16.

We also measured the efficiency with which each version of the Performance Consultant performs its search. An efficient performance tool will perform its search while inserting a minimum amount of instrumentation into the application. Table 3 also shows the number of mini-trampolines used by the two search methods, each of which corresponds to the insertion of a single instrumentation primitive. The new version of the Performance Consultant can be seen to provide a dramatic improvement in terms of efficiency. The number of mini-trampolines used by the previous version of the PC is more than an order of magnitude larger than used by the new PC for both go and Draco, and also significantly larger for the other applications studied. This improvement in efficiency results in less perturbation of the application and therefore a greater degree of accuracy in performance diagnosis.

Although the callgraph-based performance consultant identifies a greater number of bottlenecks than the original version of the of the performance consultant, it suffers one drawback that stems from the use of inclusive metrics. Inclusive timing metrics collect data specific to one function and all of its callees. Because the performance data collected is not restricted to a single function, it is difficult to evaluate a particular function in isolation and determine its exigency. For example, only 13% of those functions determined bottlenecks by the callgraph-based performance consultant are truly bottlenecks. The remainder are functions which have been classified bottlenecks en route to the discovery of true application bottlenecks. One solution to this inclusive bottleneck ambiguity is to re-evaluate all inclusive bottlenecks using exclusive metrics. Work is currently underway within the Paradyn group to implement this inclusive bottleneck verification.

6 Conclusions

We found the new callgraph-based bottleneck search in Paradyn's Performance Consultant, combined with dynamic call site instrumentation, to be much more efficient in identifying bottlenecks than its predecessor. It works faster and with less instrumentation, resulting in lower perturbation of the application and consequently greater accuracy in performance diagnosis.

Along with its advantages, the callgraph-based search has some disadvantages that remain to be addressed. Foremost among them are blind spots, where exigent functions are masked in the callgraph by their multiple parent functions, none of which themselves meet the threshold criteria to be found a bottleneck. This circumstance appears to be sufficiently rare that we have not encountered any instances yet in practice. It is also necessary to consider when and how it is most appropriate for function exigency consideration to progress from using the weak inclusive criteria to the strong exclusive criteria that determine true bottlenecks. The exclusive 'refinement' of a function found exigent using inclusive criteria can be considered as a follow-on equivalent to refine-

ment to its children, or as a reconsideration of its own exigency using the stronger criteria. Additionally, it remains to be determined how the implicit equivalence of the main program routine and 'Code' (the root of the Code hierarchy) as resource specifiers can be exploited for the most efficient searches and insightful presentation.

Acknowledgements. Matthew Cheyney implemented the initial version of the static callgraph structure. This paper benefited from the hard work of the members of the Paradyn research group. While everyone in the group influenced and helped with the results in this paper, we would like to specially thank Andrew Bernat for his support of the AIX/SP2 measurements, and Chris Chambreau for his support of the IRIX/MIPS measurements. We are grateful to the Laboratory for Laser Energetics at the University of Rochester for the use of their SGI Origin system for some of our experiments, and the various authors of the codes made available to use. Ariel Tamches provided constructive comments on the early drafts of the paper.

References

[1] W. Williams, T. Hoel, and D. Pase, "The MPP Apprentice performance tool: Delivering the performance of the Cray T3D", in **Programming Environments for Massively Parallel Distributed Systems**, K.M. Decker and R.M. Rehmann, editors, Birkhäuser, 1994.

[2] A. Beguelin, J. Dongarra, A. Geist, and V.S. Sunderam, "Visualization and Debugging in a Heterogeneous Environment", *IEEE Computer* **26**, 6, June 1993.

[3] H.M. Gerndt and A. Krumme, "A Rule-Based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems", *2nd Int'l Workshop on High-Level Programming Models and Supportive Environments*, Genève, Switzerland, April 1997.

[4] J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", 7th *Int'l Conf. on Supercomputing*, Tokyo, Japan, July 1993.

[5] J.K. Hollingsworth, B.P. Miller and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools", *Scalable High Performance Computing Conf.,* Knoxville, Tennessee, May 1994.

[6] J.K Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naìm, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *6th Int'l Conf. on Parallel Architectures and Compilation Techniques*, San Francisco, California, Nov. 1997

[7] K.L. Karavanic and B.P. Miller, "Improving Online Performance Diagnosis by the Use of Historical Performance Data", *SC'99*, Portland, Oregon, November 1999.

[8] J. Kohn and W. Williams, "ATExpert", *Journal of Parallel and Distributed Computing* **18**, 205–222, June 1993.

[9] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", *IEEE Computer* **28**, 11, pp. 37-46, November 1995.

[10] N. Mukhopadhyay (Mukerjee), G.D. Riley, and J.R. Gurd, "FINESSE: A Prototype Feedback-Guided Performance Enhancement System", *8th Euromicro Workshop on Parallel and Distributed Processing*, Rhodos, Greece, January 2000.

[11] Z. Xu, B.P. Miller, and O. Naìm, "Dynamic Instrumentation of Threaded Applications", *7th ACM Symp. on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.

Automatic Performance Analysis of MPI Applications Based on Event Traces

Felix Wolf and Bernd Mohr

Research Centre Jülich,
Central Institute for Applied Mathematics,
52425 Jülich, Germany,
{f.wolf, b.mohr}@fz-juelich.de

Abstract. This article presents a class library for detecting typical performance problems in event traces of MPI applications. The library is implemented using the powerful high-level trace analysis language EARL and is embedded in the extensible tool component EXPERT described in this paper. One essential feature of EXPERT is a flexible plug-in mechanism which allows the user to easily integrate performance problem descriptions specific to a distinct parallel application without modifying the tool component.

1 Introduction

The development of fast and scalable parallel applications is still a very complex and expensive process. The complexity of current systems involves incremental performance tuning through successive observations and code refinements. A critical step in this procedure is transforming the collected data into a useful hypothesis about inefficient program behavior. Automatically detecting and classifying performance problems would accelerate this process considerably.

The performance problems considered here are divided into two classes. The first is the class of well-known and frequently occurring bottlenecks which have been collected by the ESPRIT IV *Working Group on Automatic Performance Analysis: Resources and Tools* (APART) [4]. The second is the class of application specific bottlenecks which only can be specified by the application designers themselves.

Within the framework defined in the KOJAK project [6] (*Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks*) at the Research Centre Jülich which is aimed at providing a generic environment for automatic performance analysis, we implemented a class library capable of identifying typical bottlenecks in event traces of MPI applications.

The class library uses the high-level trace analysis language EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage) [11] as foundation and is incorporated in an extensible and modular tool architecture called EXPERT (**Ex**tensible **Per**formance **T**ool) presented in this article. To support the easy integration

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 123–132, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of application-specific bottlenecks, EXPERT provides a flexible *plug-in* mechanism which is capable of handling an arbitrary set of performance problems specified in the EARL language.

First, we summarize the EARL language together with the EARL model of an event trace in the next section. In section 3 we present the EXPERT tool architecture and its extensibility mechanism in more detail. Section 4 describes the class library for detection of typical MPI performance problems which is embedded in EXPERT. Applying the library to a real application in section 5 shows how our approach can help to understand the performance behavior of a parallel program. Section 6 discusses related work and section 7 concludes the paper.

2 EARL

In the context of the EARL language a performance bottleneck is considered as an event pattern or compound event which has to be detected in the event trace after program termination. The compound event is build from primitive events such as those associated with entering a program region or sending a message. The pattern can be specified as a script containing an appropriate search algorithm written in the EARL trace analysis language. The level of abstraction provided by EARL allows the algorithm to have a very simple structure even in case of complex event patterns.

A performance analysis script written in EARL usually takes one or more trace files as input and is then executed by the EARL interpreter. The input files are automatically mapped to the EARL event trace model, independently of the underlying trace format, thereby allowing efficient and portable random access to the events recorded in the file. Currently, EARL supports the VAMPIR [1], ALOG, and CLOG [7] trace formats.

2.1 The EARL Event Trace Model

The EARL event trace model defines the way an EARL programmer views an event trace. It describes event types and system states and how they are related. An event trace is considered as a sequence of events. The events are numbered according to their chronological position within the event trace. EARL provides four predefined event types: entering (named *enter*) and leaving (*exit*) a code region of the program, and sending (*send*) as well as receiving (*recv*) a message. In addition to these four standard event types the EARL event trace model provides a template without predefined semantics for event types that are not part of the basic model. If supported by the trace format, regions may be organized in groups (e.g. user or communication functions).

The event types share a set of typical attributes like a timestamp (*time*) and the location (*loc*) where the event happened. The event type is explicitly given as a string attribute (*type*). However, the most important attribute is the

position (*pos*) which is needed to uniquely identify an event and which is assigned according to the chronological order within the event trace.

The *enter* and *exit* event types have an additional *region* attribute specifying the name of the region entered or left. *send* and *recv* have attributes describing the destination (*dest*), source (*src*), tag (*tag*), length (*len*), and communicator (*com*) of the message.

The concepts of region instances and messages are realized by two special attributes. The *enterptr* attribute which is common to all event types points to the *enter* event that determines the region instance in which the event happened. In particular *enterptr* links two matching *enter* and *exit* events together. Apart from that, *recv* events provide an additional *sendptr* attribute to identify the corresponding *send* event.

For each position in the event trace, EARL also defines a system state which reflects the state after the event at this position took place. A system state consists of a region stack per location and a message queue. The region stack is defined as the set of *enter* events that determine the region instances in which the program executes at a given moment, and the message queue is defined as the set of *send* events of the messages sent but not yet received at that time.

2.2 The EARL Language

The core of the current EARL version is implemented as C++ classes whose interfaces are embedded in each of the three popular scripting languages Perl, Python, and Tcl. However, in the remainder of this article we refer only to the Python mapping.

The most important class is named `EventTrace` and provides a mapping of the events from a trace file to the EARL event trace model. `EventTrace` offers several operations for accessing events: The operation `event()` returns a hash value, e.g. a Python dictionary. This allows to access individual attributes by providing the attribute name as hash key. Alternatively, you can get a literal representation of an event, e.g. in order to write some events to a file.

EARL automatically calculates the state of the region stacks and the message queue for a given event. The `stack()` operation returns the stack of a specified location represented as a list containing the positions of the corresponding *enter* events. The `queue()` operation returns the message queue represented as a list containing the positions of the corresponding *send* events. If only messages with a certain source or destination are required, their locations can by specified as arguments to the `queue()` operation.

There are also several operations to access general information about the event trace, e.g. to get the number of locations used by a parallel application.

For a complete description of the EARL language we refer to [12].

3 An Extensible and Modular Tool Architecture

The EXPERT tool component for detection of performance problems in MPI applications is implemented in Python on top of EARL. It is designed according

to the specifications and terminology presented in [4]. There, an *experiment* which is represented by the performance data collected during one program run, i.e. a trace file in our case, is characterized by the occurrence of different *performance properties*. A performance property corresponds to one aspect of inefficient program behavior. The existence of a property can be checked by evaluating appropriate conditions based on the events in the trace file.

The architecture of the trace analysis tool EXPERT is mainly based on the idea of separating the performance analysis process from the definitions of the performance properties we are looking for. Performance properties are specified as Python classes which represent patterns to be matched against the event trace and which are implemented using the EARL language. Each pattern provides a *confidence* attribute indicating the confidence of the assumption made by a successful pattern match about the occurrence of a performance property. The *severity* attribute gives information about the importance of the property in relation to other properties.

All pattern classes provide a common interface to the tool. As long as these classes fulfill the contract stated by the common interface, EXPERT is able to handle an arbitrary set of patterns.

The user of EXPERT interactively selects a subset of the patterns offered by the tool by clicking the corresponding checkbuttons on the graphical user interface. Activating a pattern triggers a pattern specific configuration dialogue during which the user can set different parameters if necessary. Optionally, he can choose a program region to concentrate the analysis process only on parts of the parallel application.

The actual trace analysis performed by EXPERT follows an event driven approach. First there is some initialization for each of the selected patterns which are represented by instances of the corresponding classes. Then the tool starts a thread which walks sequentially through the trace file and for each single event invokes a callback function provided by the pattern object according to the type of the event. The callback function itself may request additional events, e.g. when it follows a link emanating from the current event which is passed as an argument, or query system state information by calling appropriate EARL commands. After the last event has been reached, EXPERT applies a wrapup operation to each pattern object which calculates result values based on the data collected during the walk through the trace. Based on these result values the severity of the pattern is computed. Furthermore, each pattern may provide individual results, e.g. concerning the execution phase of the parallel program in which a pattern match was found.

Customizing EXPERT with Plug-Ins

The signature of the operations provided by the pattern classes is defined in a common base class *Pattern*, but each derived class may provide an individual implementation.

EXPERT currently manages two sets of patterns, i.e. one set of patterns representing frequently occurring message passing bottlenecks which is described

in the next section and one set of user defined patterns which may be used to detect performance problems specific to a distinct parallel application.

If users of EXPERT want to provide their own pattern, they simply write another realization (subclass) of the *Pattern* interface (base class). Now, all they have to do is to insert the new class definition in a special file which implements a *plug-in* module. At startup time EXPERT dynamically queries the module's namespace and looks for all subclasses of *Pattern* from which it is now able to build instances without knowing the number and names of all new patterns in advance. By providing its own configuration dialogue which may be launched by invoking a configure operation on it each pattern can be seamlessly integrated into the graphical user interface.

4 Automatic Performance Analysis of MPI Programs

Most of the patterns for detection of typical MPI performance properties we implemented so far correspond to MPI related specifications from [4]. The set of patterns is split into two parts. The first part is mainly based on summary information, e.g. involving the total execution times of special MPI routines which could also be provided by a profiling tool. However, the second part involves idle times that can only be determined by comparing the chronological relation between concrete region instances in detail. This is where our approach can demonstrate its full power. A major advantage of EXPERT lies in its ability to handle both groups of performance properties in one step.

Currently, EXPERT supports the following performance properties:[1]

Communication costs: The severity of this property represents the time used for communication over all participating processes, i.e. the time spent in MPI routines except for those that perform synchronization only. The computed amount of time is returned as severity.

Synchronization costs: The time used exclusively for synchronization.

IO costs: The time spent in IO operations. It is essential for this property that all IO routines can be identified by their membership in an IO group whose name can be set as a parameter in the configuration dialogue.

Costs: The severity of this property is simply the sum of the previous three properties. Note that while the severities of the individual properties above may seem uncritical, the sum of all together may be considered as a performance problem.

Dominating communication: This property denotes the costs caused by the communication operation with maximum execution time relative to other communication operations. Besides the total execution time (severity) the name of the operation can also be requested.

[1] In [4] the severity specification is based on a scaling factor which represents the value to which the original value should be compared. In EXPERT this scaling factor is provided by the tool and is not part of the pattern specification. Currently it is the inverse of the total execution time of the program region being investigated.

Frequent communication: A program region has the property *frequent communication*, if the average execution time of communication statements lies below a user defined threshold. The severity is defined as the costs caused by those statements. Their names are also returned.

Big messages: A program region has the property *big messages*, if the average length of messages sent or received by some communication statements is greater than a user defined threshold. The severity is defined as the costs caused by those statements. Their names are also returned.

Uneven MP distribution: A region has this property, if communication statements exist where the standard deviation of the execution times with respect to single processes is greater than a user defined threshold multiplied with the mean execution time per process. The severity is defined as the costs caused by those statements. Their names are also returned.

Load imbalance at barrier: This property corresponds to the idle time caused by load imbalance at a barrier operation. The idle times are computed by comparing the execution times per process for each call of `MPI_BARRIER`. To work correctly the implementation of this property requires all processes to be involved in each call of the collective barrier operation. The severity is just the sum of all measured idle times.

Late sender: This property refers to the amount of time wasted, when a call to `MPI_RECV` is posted before the corresponding `MPI_SEND` is executed. The idle time is measured and returned as severity. We will look at this pattern in more detail later.

Late receiver: This property refers to the inverse case. A `MPI_SEND` blocks until the corresponding receive operation is called. This can happen for several reasons. Either the implementation is working in synchronous mode by default or the size of the message to be sent exceeds the available buffer space and the operation blocks until the data is transfered to the receiver. The behavior is similar to an `MPI_SSEND` waiting for message delivery. The idle time is measured and the sum of all idle times is returned as severity value.

Slow slaves: This property refers to the *master-slave* paradigm and identifies a situation where the master waits for results instead of doing useful work. It is a specialization of the *late sender* property. Here only messages sent to a distinct *master* location which can be supplied as a parameter are considered.

Overloaded master: If the slaves have to wait for new tasks or for the master to receive the results of finished tasks, this property can be observed. It is implemented as a mix of *late sender* and *late receiver* again involving a special *master* location.

Receiving messages in wrong order: This property which has been motivated by [8] deals with the problem of passing messages *out of order*. The sender is sending messages in a certain order, but the receiver is expecting the arrival in another order. The implementation locates such situations by querying the message queue each time a message is received and looking for *older* messages with the same target as the current message. Here, the severity is defined as the costs resulting from all communication operations involved in such situations.

Whereas the first four properties serve more as an indication that a performance problem exists, the latter properties reveal important information about the reason for inefficient program behavior. Note that especially the implementations of the last six properties require the detection of quite complex event patterns and therefore can benefit from the powerful services provided by the EARL interpreter.

5 Analyzing a Real Application

In order to demonstrate how the performance analysis environment presented in the previous sections can be used to gain deeper insight into performance behavior we consider a real application named *CX3D* which is used to simulate the *Czochralski* crystal growth [9], a method being applied in the silicon waver production. The simulation covers the convection processes occurring in a rotating cylindric crucible filled with liquid melt.

The convection which strongly influences the chemical and physical properties of the growing crystal is described by a system of partial differential equations. The crucible is modeled as a three dimensional cubical mesh with its round shape being expressed by cyclic border conditions. The mesh is distributed across the available processes using a two dimensional spatial decomposition. Most of the execution time is spent in a routine called `VELO` and is used to calculate the new velocity vectors. Communication is required when the computation involves mesh cells from the border of each processors' sub-domain.

The `VELO` routine has been investigated with respect to the *late sender* pattern. This pattern determines the time between the calls of two corresponding point-to-point communication operations which involves identifying the matching *send* and *recv* events. The Python class definition of the pattern is presented in Fig. 1.

Each time EXPERT encounters a *recv* event the `recv_callback()` operation is invoked on the pattern instance and a dictionary containing the *recv* event is passed as an argument. The pattern first tries to locate the *enter* event of the enclosing region instance by following the *enterptr* attribute. Then, the corresponding *send* event is determined by tracing back the *sendptr* attribute. Now, the pattern looks for the *enter* event of the region instance from which the message originated. Next, the chronological difference between the two *enter* events is computed. Since the `MPI_RECV` has to be posted earlier than the `MPI_SEND`, the `idle_time` has to be greater than zero. Last, we check whether the analyzed region instances really belong to `MPI_SEND` and `MPI_RECV` and not to e.g. `MPI_BCAST`. If all of that is true, we can add the measured idle time to the global sum `self.sum_idle_time`. The complete pattern class as contained in the EXPERT tool also computes the distribution of the losses introduced by that situation across the different processes, but this is not shown in the script example.

The execution configuration of CX3D is determined by the number of processes in each of the two decomposed dimensions. The application has been

```
class LateSender(Pattern):
    [... initialization operations ...]
    def recv_callback(self, recv):
        recv_start = self.trace.event(recv['enterptr'])
        send = self.trace.event(recv['sendptr'])
        send_start = self.trace.event(send['enterptr'])
        idle_time =  send_start['time'] - recv_start['time']
        if (idle_time > 0 and
            send_start['region'] == "MPI_SEND" and
            recv_start['region'] == "MPI_RECV"):
            self.sum_idle_time = self.sum_idle_time + idle_time
    def confidence(self):
        return 1                  # safe criterion
    def severity(self):
        return self.sum_idle_time
```

Fig. 1. Python class definition of the *late sender* pattern

executed using different configurations on a Cray T3E. The results are shown in Table 1. The third column shows the fraction (severity) of execution time spent in communication routines and the rightmost column shows the fraction (severity) of execution time lost by late sender. The results indicate that the process topology has major impact on the communication costs. This effect is to a significant extent caused by the late sender pattern.

For example, in the 8 x 1 configuration the last process is assigned only a minor portion of the total number of mesh cells since the corresponding mesh dimension length is not divisible by 8. This load imbalance is reflected in the calculated distribution of the losses introduced by the pattern (Table 2).

Table 1. Idle times in routine VELO introduced by *late sender*

#Processes	Configuration	Communication Cost	Late Sender
8	2 x 4	0.191	0.050
8	4 x 2	0.147	0.028
8	8 x 1	0.154	0.035
16	4 x 4	0.265	0.055
16	8 x 2	0.228	0.043
16	16 x 1	0.211	0.030
32	8 x 4	0.335	0.063
32	16 x 2	0.297	0.035

However, the results produced by the remaining configurations may be determined by other effects as well.

Table 2. Distribution of idle times in an 8 x 1 configuration

Process	0	1	2	3	4	5	6	7
Fraction	0.17	0.08	0.01	0.01	0.01	0.01	0.05	0.68

6 Related Work

An alternative approach to describe complex event patterns was realized by [2]. The proposed Event Definition Language (EDL) allows the definition of compound events in a declarative manner based on extended regular expressions where primitive events are clustered to higher-level events by certain formation operators. Relational expressions over the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. However, problems arise when trying to describe events that are associated with some kind of state.

KAPPA-PI [3] performs automatic trace analysis of PVM programs based on a set of predefined rules representing common performance problems. It also demonstrates, how modern scripting technology, i.e. Perl in this case, can be used to implement valuable tools.

The specifications from [4] on top of which the class library presented in this paper is build, serve also as foundation for a profiling based tool COSY [5]. Here the performance data is stored in a relational database and the performance properties are represented by appropriate SQL queries.

A well-known tool for automatic performance analysis is developed in the Paradyn project [10]. In contrast to our approach Paradyn uses online instrumentation. A predefined set of bottleneck hypotheses based on metrics described in a dedicated language is used to prove the occurrence of performance problems.

7 Conclusion and Future Work

In this article we demonstrated how the powerful services offered by the EARL language can be made available to the designer of a parallel application by providing a class library for the detection of typical problems affecting the performance of MPI programs.

The class library is incorporated into EXPERT, an extensible tool component which is characterized by a separation of the performance problem specifications from the actual analysis process. This separation enables EXPERT to handle an arbitrary set of performance problems.

A graphical user interface makes utilizing the class library for detection of typical MPI performance problems straightforward. In addition, a flexible plug-in mechanism allows the experienced user to easily integrate problem descriptions specific to a distinct parallel application without modifying the tool.

Whereas our first prototype realizes only a simple concept of selecting a search focus, we want to integrate a more elaborate hierarchical concept supporting stepwise refinements and experiment management in later versions.

Furthermore, we intend to support additional programming paradigms like *shared memory* and in particular hybrid models in the context of SMP cluster computing. A first step would be to extend the EARL language towards a broader set of event types and system states associated with such paradigms.

References

[1] A. Arnold, U. Detert, and W.E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[2] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, February 1886.

[3] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Evaluation of Parallel Programs. In *Proc. of the 6th Euromicro Workshop on Parallel and Disributed Pocessing(PDP'98)*, 1998.

[4] T. Fahringer, M. Gerndt, and G. Riley. Knowledge Specification for Automatic Performance Analysis. Technical report, ESPRIT IV Working Group APART, 1999.

[5] M. Gerndt and H.-G. Eßer. Specification Techniques for Automatic Performance Analysis Tools. In *Proceedings of the 5th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2000), in conjunction with IPDPS 2000*, Cancun, Mexico, May 2000.

[6] M. Gerndt, B. Mohr, M. Pantano, and F. Wolf. Performance Analysis for CRAY T3E. In IEEE Computer Society, editor, *Proc. of the 7th Euromicro Workshop on Parallel and Disributed Pocessing(PDP'99)*, pages 241–248, 1999.

[7] W. Gropp and E. Lusk. *User's Guide for MPE: Extensions for MPI Programs.* Argonne National Laboratory, 1998. http://www-unix.mcs.anl.gov/mpi/mpich/.

[8] J.K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, Oktober 1996.

[9] M. Mihelcic, H. Wenzl, and H. Wingerath. Flow in Czochralski Crystal Growth Melts. Technical Report Jül-2697, Research Centre Jülich, December 1992.

[10] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[11] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In A. Hoekstra and B. Hertzberger, editors, *Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, pages 503–512, Amsterdam (The Netherlands), 1999.

[12] F. Wolf and B. Mohr. EARL - Language Reference. Technical Report ZAM-IB-2000-01, Research Centre Jülich, Germany, February 2000.

Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions

Jacques Chassin de Kergommeaux[1] and Benhur de Oliveira Stein[2]

[1] ID-IMAG, ENSIMAG - antenne de Montbonnot, ZIRST, 51, avenue Jean Kuntzmann, 38330 MONTBONNOT SAINT MARTIN, France.
Jacques.Chassin-de-Kergommeaux@imag.fr
http://www-apache.imag.fr/∼chassin

[2] Departamento de Eletrônica e Computação, Universidade Federal de Santa Maria, Brazil.
benhur@inf.UFSM.br
http://www.inf.ufsm.br/∼benhur

Abstract. Pajé is an interactive visualization tool for displaying the execution of parallel applications where a (potentially) large number of communicating threads of various life-times execute on each node of a distributed memory parallel system. The main novelty of Pajé is an original combination of three of the most desirable properties of visualization tools for parallel programs: extensibility, interactivity and scalability. This article mainly focuses on the extensibility property of Pajé, ability to easily add new functionalities to the tool. Pajé was designed as a data-flow graph of modular components to ease the replacement of existing modules or the implementation of new ones. In addition the genericity of Pajé allows application programmers to tailor the visualization to their needs, by simply adding tracing orders to the programs being traced.
Keywords: performance debugging, visualization, MPI, pthread, parallel programming.

1 Introduction

The Pajé visualization tool was designed to allow programmers to visualize the executions of parallel programs using a potentially large number of communicating threads (lightweight processes) evolving dynamically. The visualization of the executions is an essential tool to help tuning applications using such a parallel programming model.

Visualizing a large number of threads raises a number of problems such as coping with the lack of space available on the screen to visualize them and understanding such a complex display. The graphical displays of most existing visualization tools for parallel programs [8, 9, 10, 11, 14, 15, 16] show the activity of a fixed number of nodes and inter-nodes communications; it is only possible to represent the activity of a single thread of control on each of the nodes. Some tools were designed to display multi-threaded programs [7, 17]. However, they support a programming model involving a single level of parallelism within a node, this node being in general a shared-memory multiprocessor. Our programs execute on several nodes: within the same node, threads communicate using synchronization primitives; however, threads executing on different nodes communicate by message passing.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 133–140, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The most innovative feature of Pajé is to combine the characteristics of interactivity and scalability with extensibility. In contrast with passive visualization tools [8, 14] where parallel program entities — communications, changes in processor states, etc. — are displayed as soon as produced and cannot be interrogated, it is possible to inspect all the objects displayed in the current screen and to move back in time, displaying past objects again. Scalability is the ability to cope with a large number of threads. Extensibility is an important characteristic of visualization tools to cope with the evolution of parallel programming interfaces and visualization techniques. Extensibility gives the possibility to extend the environment with new functionalities: processing of new types of traces, adding new graphical displays, visualizing new programming models, etc.

The interactivity and scalability characteristics of Pajé were described elsewhere [2, 4]. This article focuses on the extensibility characteristics: modular design easing the addition of new modules, semantics independent modules which allow them to be used in a large variety of contexts and especially genericity of the simulator component of Pajé which gives to application programmers the ability to define what they want to visualize and how it must be done.

The main functionalities of Pajé are summarized in the next section. The following section describes the extensibility of Pajé before the conclusion.

2 Outline of Pajé

Pajé was originally designed to ease performance debugging of ATHAPASCAN programs by visualizing their executions and because no existing visualization tool could be used to visualize such multi-threaded programs.

2.1 ATHAPASCAN: A Thread-Based Parallel Programming Model

Combining threads and communications is increasingly used to program irregular applications, mask communications or I/O latencies, avoid communication deadlocks, exploit shared-memory parallelism and implement remote memory accesses [5, 6]. The ATHAPASCAN [1] programming model was designed for parallel hardware systems composed of shared-memory multi-processor nodes connected by a communication network. Inter-nodes parallelism is exploited by a fixed number of system-level processes while inner parallelism, within each of the nodes, is implemented by a network of communicating threads evolving dynamically. The main functionalities of ATHAPASCAN are dynamic local or remote thread creation and termination, sharing of memory space between the threads of the same node which can synchronize using locks or semaphores, and blocking or non-blocking message-passing communications between non local threads, using ports. Combining the main functionalities of MPI [13] with those of `pthread` compliant libraries, ATHAPASCAN can be seen as a "thread aware" implementation of MPI.

2.2 Tracing of Parallel Programs

Execution traces are collected during an execution of the observed application, using an instrumented version of the ATHAPASCAN library. A non-intrusive, statistical method

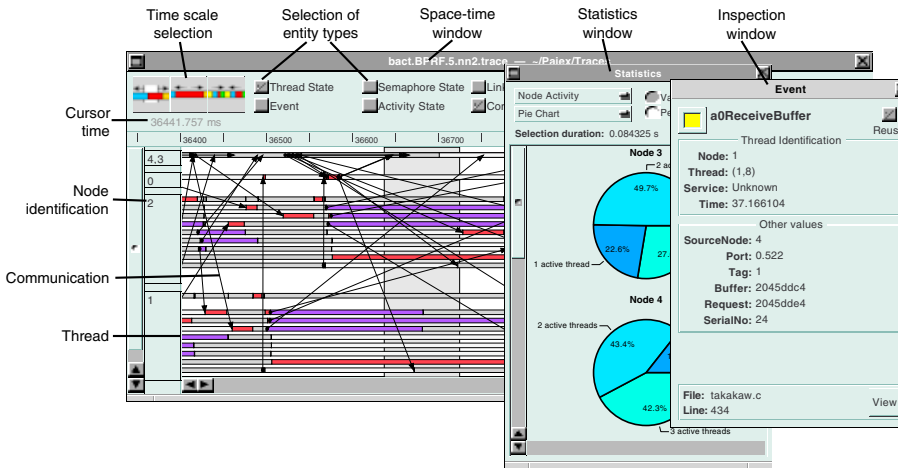


Fig. 1. Visualization of an ATHAPASCAN program execution
Blocked thread states are represented in clear color; runnable states in a dark color. The smaller window shows the inspection of an event.

is used to estimate a precise global time reference [12]. The events are stored in local event buffers, which are flushed when full to local event files. Recorded events may contain source code information in order to implement source code click-back — from visualization to source code — and click-forward — from source code to visualization — in Pajé.

2.3 Visualization of Threads in Pajé

The visualization of the activity of multi-threaded nodes is mainly performed in a diagram combining in a single representation the states and communications of each thread (see figure 1) . The horizontal axis represents time while threads are displayed along the vertical axis, grouped by node. The space allocated to each node of the parallel system is dynamically adjusted to the number of threads being executed on this node. Communications are represented by arrows while the states of threads are displayed by rectangles. Colors are used to indicate either the type of a communication, or the activity of a thread.

The states of semaphores and locks are represented like the states of threads: each possible state is associated with a color, and a rectangle of this color is shown in a position corresponding to the period of time when the semaphore was in this state. Each lock is associated with a color, and a rectangle of this color is drawn close to the thread that holds it. Moving the mouse pointer over the representation of a blocked thread state highlights the corresponding semaphore state, allowing an immediate recognition. Similarly, all threads blocked in a semaphore are highlighted when the pointer is moved over the corresponding state of the semaphore.

In addition, Pajé offers many possible interactions to programmers: displayed objects can be inspected to obtain all the information available for them (see inspection

window in figure 1), identify related objects or check the corresponding source code. Selecting a line in the source code browser highlights the events that have been generated by this line.

Progress of the simulation is entirely driven by user-controlled time displacements: at any time during a simulation, it is possible to move forward or backward in time. Memory usage is kept to acceptable levels by a mechanism of checkpointing the internal state of the simulator and re-simulating when needed.

It is not possible to represent simultaneously all the information that can be deduced from the execution traces. Pajé offers several filtering and zooming functionalities to help programmers cope with this large amount of information to give users a simplified, abstract view of the data. Figure 1 exemplifies one of the filtering facilities provided by Pajé where the top most line represents the number of active threads of a group of two nodes (nodes 3 and 4) and a pie graph the CPU activity in the time slice selected in the space-time diagram (see [2, 3] for more details).

3 Extensibility

Extensibility is a key property of a visualization tool. The main reason is that a visualization tool being a very complex piece of software, costly to implement, its lifetime ought to be as long as possible. This will be possible only if the tool can cope with the evolutions of parallel programming models and of the visualization techniques, since both domains are evolving rapidly. Several characteristics of Pajé were designed to provide a high degree of extensibility: modular architecture, flexibility of the visualization modules and genericity of the simulation module.

3.1 Modular Architecture

To favor extensibility, the architecture of Pajé is a data flow graph of software modules or components. It is therefore possible to add a new visualization component or adapt to a change of trace format by changing the trace reader component without changing the remaining of the environment. This architectural choice was inspired by Pablo [14], although the graph of Pajé is not purely data-flow for interactivity reasons: it also includes control-flow information, generated by the visualization modules to process user interactions and trigger the flow of data in the graph (see [2, 3] for more details).

3.2 Flexibility of Visualization Modules

The Pajé visualization components do not depend on specific parallel programming models. Prior to any visualization they receive as input the description of the types of the objects to be visualized as well as the relations between these objects and the way these objects ought to be visualized (see figure 2). The only constraints are the hierarchical nature of the type relations between the visualized objects and the ability to place each of these objects on the time-scale of the visualization. The hierarchical type description is used by the visualization components to query objects from the preceding components in the graph.

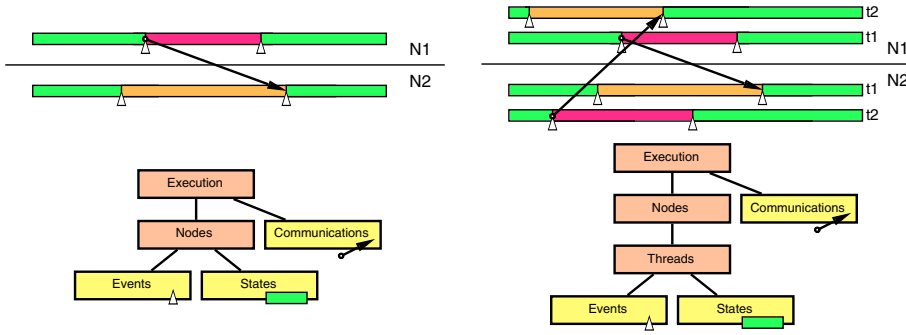


Fig. 2. Use of a simple type hierarchy
The type hierarchy on the left-hand side of the figure defines the type hierarchical relations between the objects to be visualized and how these objects should be represented: communications as arrows, thread events as triangles and thread states as rectangles. The right-hand side shows the changes necessary to the hierarchy in order to represent threads.

This type description can be changed to adapt to a new programming model (see section 3.3) or during a visualization, to change the visual representation of an object upon request from the user. This feature is also used by the filtering components when they are dynamically inserted in the data-flow graph of Pajé — for example between the simulation and visualization components to zoom from a detailed visualization and obtain a more global view of the program execution (see [2, 3] for more details).

The type hierarchies used in Pajé are trees whose leaves are called *entities* and intermediate nodes *containers*. Entities are elementary objects such as events, thread states or communications. Containers are higher level objects, including entities or lower level containers (see figure 2). For example: all events occurring in thread 1 of node 0 belong to the container "thread-1-of-node-0".

3.3 Genericity of Pajé

The modular structure of Pajé as well as the fact that filter and visualization components are independent of any programming model makes it "easy" for tool developers to add a new component or extend an existing one. These characteristics alone would not be sufficient to use Pajé to visualize various programming models if the simulation component were dependent on the programming model: visualizing a new programming model would then require to develop a new simulation component, which is still an important programming effort, reserved to experienced tool developers.

On the contrary, the generic property of Pajé allows application programmers to define *what* they would like to visualize and *how* the visualized objects should be represented by Pajé. Instead of being computed by a simulation component, designed for a specific programming model such as ATHAPASCAN, the type hierarchy of the visualized objects (see section 3.2) can be defined by the application programmer, by inserting several definitions and commands *in the application program to be traced and visualized.* These definitions and commands are collected by the tracer (see section 2.2) so that they can be passed to the Pajé simulation component. The simulator uses these

Table 1. Containers and entities types definitions and creation

Type definition	Creation of entities
pajeDefineUserContainerType	pajeCreateUserContainer pajeDestroyUserContainer
pajeDefineUserEventType	pajeUserEvent
pajeDefineUserStateType	pajeSetUserState pajePushUserState pajePopUserState
pajeDefineUserLinkType	pajeStartUserLink pajeEndUserLink
pajeDefineUserVariableType	pajeSetUserVariable pajeAddUserVariable

definitions to build a new data type tree relating the objects to be displayed, this tree being passed to the following modules of the data flow graph: filters and visualization components.

New Data Types Definition. One function call is available to create new types of containers while four can be used to create new types of entities which can be events, states, links and variables. An "event" is an entity representing an instantaneous action. "States" of interest are those of containers. A "link" represents some form of connection between a source and a destination container. A "variable" stores the temporal evolution of the successive values of a data associated with a container. Table 1 contains the function calls that can be used to define new types of containers and entities. The right-hand part of figure 2 shows the effect of adding the "threads" container to the left-hand part.

Data Generation. Several functions can be used to create containers and entities whose types are defined using the primitives of the left column of table 1. Functions of the right column of table 1 are used to create events, states (and embedded states using *Push* and *Pop*), links — each link being created by one source and one destination calls — and change the values of variables.

In the example of figure 3, a new event is generated for each change of computation phase. This event is interpreted by the Pajé simulator component to generate the corresponding container state. For example the following call indicates that the computation is entering in a "Local computation" phase:

```
pajeSetUserState ( phase_state, node, local_phase, "" );
```

The second parameter indicates the container of the state (the "node" whose computation has just been changed). The last parameter is a comment that can be visualized by Pajé. The example program of figure 3 includes the definitions and creations of entities "Computation phase", allowing the visual representation of an ATHAPASCAN program execution to be extended to represent the phases of the computation. Figure 4 shows a space-time diagram visualizing the execution of this example program with the definition of the new entities.

```
unsigned phase_state, init_phase, local_phase, global_phase;
phase_state  = pajeDefineUserStateType( A0_NODE, "Computation phase");
init_phase   = pajeNewUserEntityValue( phase_state, "Initialization");
local_phase  = pajeNewUserEntityValue( phase_state, "Local computation");
global_phase = pajeNewUserEntityValue( phase_state,"Global computation");

pajeSetUserState ( phase_state, node, init_phase, "" );
initialization();
while (!converge) {
    pajeSetUserState ( phase_state, node, local_phase, "" );
    local_computation();
    send (local_data);
    receive (remote_data);
    pajeSetUserState ( phase_state, node, global_phase, "" );
    global_computation();
}
```

Fig. 3. Simplified algorithm of the example program
Added tracing primitives are shown in bold face.

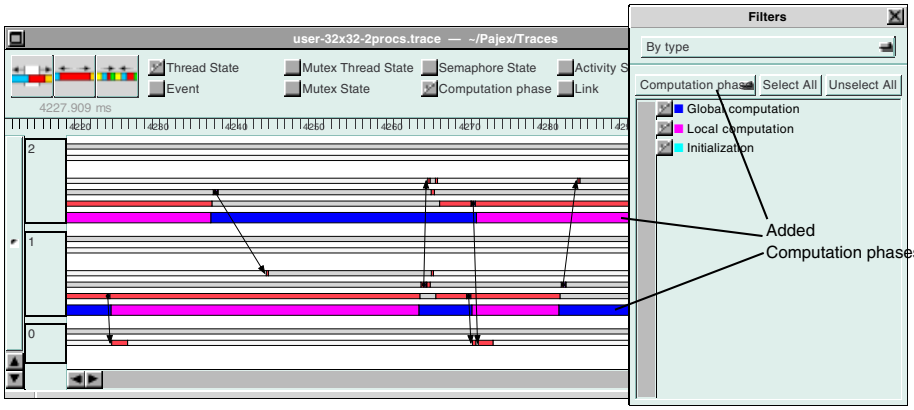


Fig. 4. Visualization of the example program

4 Conclusion

Pajé provides solutions to interactively visualize the execution of parallel applications using a varying number of threads communicating by shared memory within each node and by message passing between different nodes. The most original feature of the tool is its unique combination of extensibility, interactivity and scalability properties. Extensibility means that the tool was defined to allow tool developers to add new functionalities or extend existing ones without having to change the rest of the tool. In addition, it is possible to application programmers using the tool to define what they wish to visualize and how this should be represented. To our knowledge such a generic feature was not present in any previous visualization tool for parallel programs executions.

References

[1] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: efficiency for irregular problems. In C. Lengauer et al., editors, *EURO-PAR'97 Parallel Processing*, volume 1300 of *LNCS*, pages 591–600. Springer, Aug. 1997.

[2] J. Chassin de Kergommeaux and B. d. O. Stein. Pajé, an extensible and interactive and scalable environment for visualizing parallel program executions. Rapport de Recherche RR-3919, INRIA Rhone-Alpes, april 2000. `http://www.inria.fr/RRRT/publications-fra.html`.

[3] B. de Oliveira Stein. *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. PhD thesis, Université Joseph Fourier, Grenoble, 1999. In French. http://www-mediatheque.imag.fr.

[4] B. de Oliveira Stein and J. Chassin de Kergommeaux. Interactive visualisation environment of multi-threaded parallel programs. In *Parallel Computing: Fundamentals, Applications and New Directions*, pages 311–318. Elsevier, 1998.

[5] T. Fahringer, M. Haines, and P. Mehrotra. On the utility of threads for data parallel programming. In *Conf. proc. of the 9th Int. Conference on Supercomputing, Barcelona, Spain, 1995*, pages 51–59. ACM Press, New York, NY 10036, USA, 1995.

[6] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, Aug. 1996.

[7] K. Hammond, H. Loidl, and A. Partridge. Visualising granularity in parallel programs: A graphical winnowing system for haskell. In A. P. W. Bohm and J. T. Feo, editors, *High Performance Functional Computing*, pages 208–221, Apr. 1995.

[8] M. T. Heath. Visualizing the performance of parallel programs. *IEEE Software*, 8(4):29–39, 1991.

[9] V. Herrarte and E. Lusk. Studying parallel program behavior with upshot, 1992. http://www.mcs.anl.gov/home/lusk/upshot/upshotman/upshot.html.

[10] D. Kranzlmueller, R. Koppler, S. Grabner, and C. Holzner. Parallel program visualization with MUCH. In L. Boeszoermenyi, editor, *Third International ACPC Conference*, volume 1127 of *Lecture Notes in Computer Science*, pages 148–160. Springer Verlag, Sept. 1996.

[11] W. Krotz-Vogel and H.-C. Hoppe. The PALLAS portable parallel programming environment. In *Sec. Int. Euro-Par Conference*, volume 1124 of *Lecture Notes in Computer Science*, pages 899–906, Lyon, France, 1996. Springer Verlag.

[12] É. Maillet and C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 28:84–93, July 1995.

[13] MPI Forum. MPI: a message-passing interface standard. Technical report, University of Tennessee, Knoxville, USA, 1995.

[14] D. A. Reed et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In A. Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[15] B. Topol, J. T. Stasko, and V. Sunderam. The dual timestamping methodology for visualizing distributed applications. Technical Report GIT-CC-95-21, Georgia Institute of Technology. College of Computing, May 1995.

[16] C. E. Wu and H. Franke. *UTE User's Guide for IBM SP Systems*, 1995. http://www.research.ibm.com/people/w/wu/uteug.ps.Z.

[17] Q. A. Zhao and J. T. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, 1995.

A Statistical-Empirical Hybrid Approach to Hierarchical Memory Analysis

Xian-He Sun[1] and Kirk W. Cameron[2]

[1] Illinois Institute of Technology, Chicago IL 60616, USA
[2] Los Alamos National Laboratory, Los Alamos NM 87544, USA

Abstract. A hybrid approach that utilizes both statistical techniques and empirical methods seeks to provide more information about the performance of an application. In this paper, we present a general approach to creating hybrid models of this type. We show that for the scientific applications of interest, the scaled performance is somewhat predictable due to the regular characteristics of the measured codes. Furthermore, the resulting method encourages streamlined performance evaluation by determining which analysis steps may provide further insight to code performance.

1 Introduction

Recently statistics have provided reduction techniques for simulated data in the context of single microprocessor performance [1, 2]. Recent work has also focused on regressive techniques for studying scalability and variations in like architectures statistically with promising but somewhat limited results [3]. Generally speaking, if we were to combine the strength of such comparisons with a strong empirical or analytical technique, we could conceivably provide more information furthering the usefulness of the original model. A detailed representation of the empirical memory modeling technique we will incorporate in our hybrid approach can be found in [4].

2 The Hybrid Approach

2.1 The Hybrid Approach: Level 1

We will use *cpi*, cycles per instruction, to compare the achievable instruction-level parallelism (ILP) of particular code-machine combinations. We feel that great insight can be gathered into application and architecture performance if we break down *cpi* into contributing pieces. Following [5] and [6], we initially break *cpi* down into two parts corresponding to the pipeline and memory *cpi*.

$$cpi = cpi_{pipeline} + cpi_{memory} \tag{1}$$

Level one of the hybrid approach focuses on using two-factor factorial experiments to identify the combinations that show differences in performance that

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 141–148, 2000.
© Springer-Verlag Berlin Heidelberg 2000

warrant further investigation. Following the statistical analysis method in [3], we identify codes and machines as observations to be used in the two-factor factorial experiments. Once all measurements have been obtained, we can perform the experiments for the factors code and machine. Using statistical methods with the help of the SAS statistical tool, we gather results relating to the variations present among codes, machines and their interactions. We accomplish this via a series of hypothesis experiments where statistically we determine whether or not a hypothesis is true or false. This is the essence of the two-factor factorial experiment. This allows us to identify within a certain tolerance, the differences among code-machine combinations.

Hypothesis: Overall Effect Does Not Exist. For this experiment, the dependent variable is the overall average *cpi* measured across codes for the machines. With these parameters, disproving the hypothesis indicates that in fact, differences between the architectures for these codes exist. If this hypothesis is not disproved, then we believe with some certainty, that there are no statistical differences among the two architectures for these codes. If this hypothesis is rejected, then the next three hypotheses should be visited.

Hypothesis: Code Effect Does Not Exist. For this experiment, the dependent variable is the $cpi_{pipeline}$ term from the decoupled *cpi* of Equation 1. In practice, this term is experimentally measured when using the empirical model. If the hypothesis holds in this experiment, no difference is observed statistically for these codes on these machines at the pipeline level. Conversely, if the hypothesis is rejected, code effect does exist indicating differences at the pipeline level for this application on these architectures. In the empirical model context, if this occurs, further analysis of the $cpi_{pipeline}$ term is warranted.

Hypothesis: Machine Effect Does Not Exist. For this experiment, the dependent variable is the cpi_{memory} term from the decoupled *cpi* of Equation 1. This term can be derived experimentally as well. If the hypothesis holds in this experiment than no discernible difference between these machines statistically is apparent for these codes. Otherwise, rejecting this hypothesis indicates machine effect does exist. In the case of the empirical memory model, this warrants further investigation since it implies variations in the memory performance across code-architecture combinations.

Hypothesis: Machine-Code Interaction Does Not Exist. For this experiment, the dependent variable is overall *cpi* measured across individual codes and individual machines. If this hypothesis is held, then no machine-code interaction effects are apparent statistically. Otherwise, rejecting the hypothesis begs for further investigation of the individual codes and machines to determine why machine-code interaction changes the performance across machines. Such performance differences indicate that codes behave differently across different machines in an unexpected way, hence requiring further investigation.

2.2 The Hybrid Approach: Level 2

If Code Effect Exists, Study $cpi_{pipeline}$. This indicates fundamental differences at the on-chip architectural level. The empirical memory model does not provide insight to such performance differences, treating $cpi_{pipeline}$ as a black box. Another model, such as that found in [7] could be used to provide more insight to performance variations for such a code.

If Machine Effect Exists, Study cpi_{memory}. If machine effect exists, statistical variations are present between different codes at the memory hierarchy level across machines. This is exactly the purpose of the empirical memory model: to analyze contributions to performance from the memory hierarchy. At this point, the statistical method has provided an easy method for determining when further analysis using the memory model is necessary. This requires a more detailed look at the decoupled *cpi* in Equation 1.

Latency hiding techniques such as out-of-order execution and outstanding memory loads increase performance. We can no longer calculate overall *cpi* as simply the dot-product of maximum latencies T_i at each i level of the hierarchy and the associated number of hits to level i, h_i. We require the average latency incurred at each level in the hierarchy, t_i. Furthermore, if we define a term that expresses the ratio of average latencies to maximum latencies in terms of *cpi* we can express overall *cpi* in the following form:

$$cpi = cpi_{pipeline} + (1 - m_0) \sum_{i=2}^{nlevels} h_i * T_i \tag{2}$$

It is obvious that this is another representation of Equation 1. m_0 is formally defined as one minus the ratio of the average memory access time to the maximum memory access time:

$$m_0 = 1 - \frac{\sum_{i=2}^{nlevels} h_i * t_i}{\sum_{i=2}^{nlevels} h_i * T_i}. \tag{3}$$

m_0 quantifies the amount of overlap achieved by a processor that overlaps memory accesses with execution. $(1 - m_0)$ is the portion of time spent incurring the maximum latency.

The above equations would indicate that m_0 reflects the performance variations in *cpi* when $cpi_{pipeline}$ is constant. Calculating m_0 is costly since it requires a least square fitting first to obtain each t_i term. By applying the statistical method and through direct observation, we have isolated the conditions under which it is worthwhile to calculate the terms of Equation 2. For conditions where machine effect exists, m_0 will provide useful insight to the performance of the memory latency hiding effects mentioned. We can also use m_0 statistically to describe the scalability of a code in regard to how predictable the performance is as problem size increases. We can use other variations on the original statistical method to study the variations of m_0. This is somewhat less costly than determining m_0 for each problem size and machine combination. Nonetheless,

actually calculating m_0 values provides validation to the conclusions obtained using this technique. If m_0 values show no statistical variations or are constant as problem sizes increase, performance scales predictably and m_0 can be used for performance prediction of problem sizes not measured. If m_0 values fluctuate statistically or are not constant as problem size increases, performance does not scale predictably and cannot be used for performance prediction.

m_0 values across machines can also provide insight into performance. If statistical differences across machines for the same problem are non-existent or if $m_0 - m_0'$ is constant as problem size increases, where each m_0 represents measurements for the same code over different machines, then the memory design differences make no difference for the codes being measured.

If Machine-Code Interaction Exists, Study *cpi*. This corresponds to the fourth hypothesis. If machine-code effect exists, statistical variations are present when machine-code interactions occur. This indicates further study of the resulting *cpi* is necessary since there exist unexplained performance variations. This scenario is outside the scope of the hybrid method, but exactly what the statistical method [3] was intended to help analyze. Further focus on particular code and architecture combinations should be carried out using the statistical method.

3 Case Study

3.1 Architecture Descriptions

Single processor hierarchical memory performance is of general interest to the scientific community. For this reason, we focus on a testbed consisting of an SMP UMA architecture and a DSM NUMA architecture that share the same processing elements (the MIPS R10000 microprocessor) but differ in the implementation of the memory hierarchy. The PowerChallenge SMP and the Origin 2000 DSM machines offer state-of-the-art performance with differing implementations of the memory hierarchy.

The 200MHz MIPS R10000 microprocessor is a 4-way, out-of-order, superscalar architecture [8]. Two programmable performance counters track a number of events [9] on this chip and were a necessity for this study. Even though the R10000 processor is able to sustain four outstanding primary cache misses, external queues in the memory system of the PowerChallenge limited the actual number to less than two. In the Origin 2000, the full capability of four outstanding misses is possible. The L2 cache sizes of these two systems are also different. A processor on PowerChallenge is equipped with up to 2MB L2 cache while a CPU of Origin 2000 system always has a L2 cache of 4MB.

In our context, we are only concerned with memory hierarchy performance for a dedicated single processor. As mentioned, the PowerChallenge and Origin 2000 differ primarily in hierarchy implementation and we will not consider shared memory contributions to performance loss since all experiments have been conducted on dedicated single processors without contention for resources.

3.2 ASCI Representative Workloads

Four applications that form the building blocks for many nuclear physics simulations were used in this study. A performance comparison of the Origin and PowerChallenge architectures has been done using these codes [10] along with a detailed discussion of the codes themselves. In the interest of space, we provide only a very brief description of each code.

SWEEP is a three-dimensional discrete-ordinate transport solver. HYDRO is a two-dimensional explicit Lagrangian hydrodynamics code. HYDROT is a version of HYDRO in which most of the arrays have been transposed so that access is now largely unit-stride. HEAT solves the implicit diffusion PDE using a conjugate gradient solver for a single timestep.

3.3 Hybrid Analysis

We now apply the hybrid method to draw conclusions regarding our codes. We should note that some of the statistical steps involved can be performed by simple inspection at times. For simple cases this can be effective, but generally simple inspection will not allow quantification of the statistical variance among observations. For this reason, we utilize statistical methods in our results. Inspection should certainly be used whenever the confidence of conclusions is high. We will not present the actual numerical results of applying statistical methods to our measurements due to restrictions on space. We will however provide the general conclusions obtained via these methods, such as whether or not a hypothesis is rejected. The observations used in our experiments include various measurements for the codes mentioned at varying problem sizes. All codes were measured on both machines using the same compiled executable to avoid differences and with the following problem size constraints: HEAT [50,100], HYDRO [50,300], SWEEP [50,200], and HYDROT [50,300].

Level 1 Results For the first hypothesis, "overall effect does not exist," we use level one of the original statistical model. A straight-forward two-factor factorial experiment shows that in fact the hypothesis is rejected. This indicates further study is warranted and so, we continue with the next 3 hypotheses. Using $cpi_{pipeline}$ as the dependent variable, the two-factor factorial experiment is performed over all codes and machines to determine whether or not code effect exists. Since identical executables are used over the two machines, no variations are observed for $cpi_{pipeline}$ values over the measured codes. This is expected as the case study was prepared to focus on memory hierarchy differences. Thus the hypothesis holds, and no further study of $cpi_{pipeline}$ is warranted for these code-machine combinations.

Next, we wish to test the hypothesis "machine effect does not exist". We perform the two-factor factorial experiment using cpi_{memory}. The results show variations for the performance of cpi_{memory} across the two machines. This will require further analysis in level two of the hybrid model. Not rejecting the hypothesis would have indicated that our codes perform similarly across machines.

The third hypothesis asks whether "machine-code interaction exists". In fact, performing the two-factor factorial experiment, shows that machine-code interaction is present since we reject the hypothesis. This will have to be addressed in level two of the hybrid model as well.

Level 2 Results Now that we have addressed each of the hypothesis warranted by rejection of the "overall effect" hypothesis, we must further analyze anomalies uncovered (i.e. each rejected hypothesis). We have identified code effect existence in level 1. It is necessary to analyze the m_0 term of Equation 2. Statistical results and general inspection show strong variations with problem size in HYDRO on the Origin 2000. Less fluctuations, although significant occur for the same code on the PowerChallenge. This indicates that unpredictable variations are present in the memory performance for HYDRO. As problem size scales, the m_0 term fluctuates indicating memory accesses do not achieve a steady state to allow performance prediction for larger problem sizes. Performing the somewhat costly linear fitting required by the empirical model supports the conclusions as shown in Figures 1 and 2. In these figures, problem size represents the $y - axis$ and calculated m_0 values have been plotted. The scalability of HYDRO is in question since the rate at which latency overlap contributes to performance fluctuates.

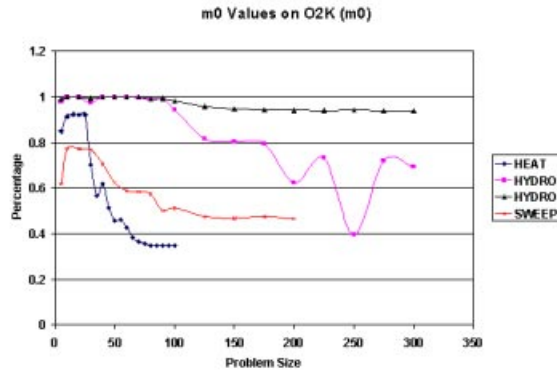


Fig. 1. m_0 values calculated on the Origin 2000.

On the other hand HEAT, HYDROT, and SWEEP show indications of predictability on the PowerChallenge. Statististical analysis of m_0 for problem sizes achieving some indication of steady state (greater than 50 for these codes - necessary to compensate for cold misses, counter accuracy, etc.) reveals little variance in m_0. For problem sizes [50,100], [75,300], and [50,200] respectively, m_0 is close to constant indicating the percentage of contribution to overlapped performance is steady. This is indicative of a code that both scales well and is somewhat predictable in nature over these machines. For these same codes on the Origin

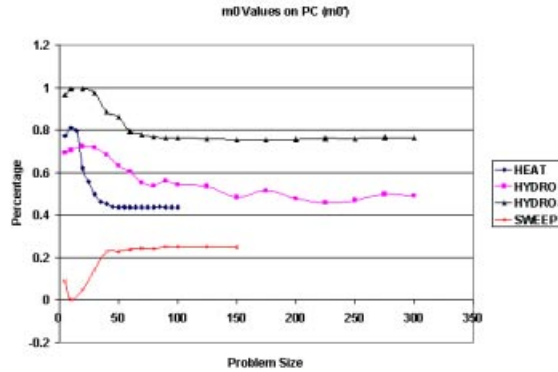


Fig. 2. m_0 values calculated on the PowerChallenge.

2000, larger problem sizes are necessary to achieve little variance in m_0. Respectively, this occurs at sizes of [75,100], [100,300], and [100,200]. The shift this time is due to the cache size difference on the Origin 2000. It takes larger problem sizes to achieve the steady state of memory behavior with respect to the latency tolerating features previously mentioned. For both machines, these three codes exhibit predictable behavior and generally good scalability.

For two codes, HEAT and HYDROT, the fluctuations in the differences between m_0 values are minimal. This can be confirmed visually in a figure not presented in this paper due to space. Such results indicate scaling between machines for these two codes over these two machines is somewhat predictable as well. HYDRO and SWEEP show larger amounts of variance for differences in m_0 values conversely. The scalability across the two machines for these codes should be analyzed further.

Finally, we must address the rejected hypothesis of machine-code interaction. Identifying this characteristic is suitable for analysis by level 2 of the original statistical method since it is not clear whether the memory architecture influence is the sole contributor to such performance variance. The statistical method refined for individual code performance [3], shows that the variance is caused by performance variations in 2 codes. Further investigation reveals that these two codes are statistically the same, allowing us to discount this rejected hypothesis.

4 Conclusions and Future Work

We have shown that the hybrid approach provides a useful analysis technique for performance evaluation of scientific codes. The technique provides insight previously not available to the stand-alone statistical method and the empirical memory model. Results indicate that 3 of the 4 codes measured show promising signs of scaled predictability. We further show that scaled performance of latency overlap is good for these same three codes. Further extensions to multi-processors

and other empirical/analytical models are future directions of this research. The authors wish to thank the referees for their suggestions regarding earlier versions of this paper. The first author was supported in part by NSF under grants ASC-9720215 and CCR-9972251.

References

[1] R. Carl and J. E. Smith, *Modeling superscalar processors via statistical simulation*, Workshop on Performance Analysis and its Impact on Design (PAID), Barcelona, Spain, 1998.

[2] D. B. Noonburg and J. P. Shen, *A framework for statistical modeling of superscalar processor performance*, 3rd International Symposium on High Performance Computer Architecture, San Antonio, TX, 1997.

[3] X. -H. Sun, D. He, K. W. Cameron, and Y. Luo, *A Factorial Performance Evaluation for Hierarchical Memory Systems*, Proceedings of IPPS/SPDP 1999, April, 1999.

[4] Y. Luo, O. M. Lubeck, H. Wasserman, F. Bassetti and K. W. Cameron, *Development and Validation of a Hierarchical Memory Model Incorporating CPU- and Memory-operation Overlap*, Proceedings of WOSP '98, October, 1998.

[5] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, Prentice Hall, pp.35-39 ,1998.

[6] P. G. Emma, *Understanding some simple processor-performance limits*, IBM Journal of Research and Development, vol. 41, 1997.

[7] K. Cameron, and Y. Luo, *Instruction-level microprocessor modeling of scientific applications*, Lecture Notes in Computer Science 1615, pp. 29–40, May 1999.

[8] K. C. Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE Micro, April, 1996, pp. 28–40.

[9] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, Performance Analysis Using the MIPS R10000 Performance Counters, Proc. Supercomputing '96, IEEE Computer Society, Los Alamitos, CA, 1996.

[10] Y. Luo, O. M. Lubeck, and H. J. Wasserman, *Preliminary Performance Study of the SGI Origin2000*, Los Alamos National Laboratory Unclassified Release LAUR 97-334, 1997.

Use of Performance Technology for the Management of Distributed Systems

Darren J. Kerbyson[1], John S. Harper[1], Efstathios Papaefstathiou[2], Daniel V. Wilcox[1], Graham R. Nudd[1]

[1]High Performance Systems Laboratory, Department of Computer Science, University of Warwick, UK
{djke,john}@dcs.warwick.ac.uk
[2]Microsoft Research, Cambridge, UK

Abstract. This paper describes a toolset, PACE, that provides detailed predictive performance information throughout the implementation and execution stages of an application. It is structured around a hierarchy of performance models that describes distributed computing systems in terms of its software, parallelisation and hardware components, providing performance information concerning expected execution time, scalability and resource use of applications. A principal aim of the work is to provide a capability for rapid calculation of relevant performance numbers without sacrificing accuracy. The predictive nature of the approach provides both pre- and post- implementation analyses, and allows implementation alternatives to be explored prior to the commitment of an application to a system. Because of the relatively fast analysis times, these techniques can be used at run-time to assist in application steering and efficient management of the available system resources.

1 Introduction

The increasing variety and complexity of high-performance computing systems requires a large number of systems issues to be assessed prior to the execution of applications on the available resources. The optimum computing configuration, the preferred software formulation, and the estimated computation time are only a few of the factors that need to be evaluated prior to making expensive commitments in hardware and software development. Furthermore, for effective evaluation the hardware system and the application software must be addressed simultaneously, resulting in an analysis problem of considerable complexity. This is particularly true for networked and distributed systems where system resource and software partitioning present additional difficulties.

The current research into GRID based computing [1] have the potential of providing access to a multitude of processing systems in a seamless fashion. That is, from a user's perspective, applications may be able to be executed on such a GRID without the need of knowing which systems are being used, or where they are physically located.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 149-159, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Such goals within the high performance community will rely on accurate performance analysis capabilities. There is a clear need to determine the best application to system resource mapping, given a number of possible choices in available systems, the current dynamic behaviour of the systems and networks, and application configurations. Such evaluations will need to be undertaken quickly so as not to impact the performance of the systems. This is analogous to current simple scheduling systems which often do not take into account the expected run-time of the applications being dealt with.

The performance technology described in this work is aimed at provided dynamic performance information on the expected run-time of applications across heterogeneous processing systems. It is based on the use of a multi-level framework encompassing all aspects of system and software. By reducing the performance calculation to a number of simple models, arbitrarily complex systems can be represented to any level of detail.

The work is part of a comprehensive effort to develop a Performance Analysis and Characterisation Environment (PACE), which will provide quantitative data concerning the performance of sophisticated applications running on high-performance systems. Because the approach does not rely on obtaining data of specific applications operating on specific machine configurations, this type of analysis provides predictive information, including:

- Execution Time
- Scalability
- On-the-fly Steering
- System Sizing
- Mapping Strategies
- Dynamic Scheduling

PACE can supply accurate performance information for both the detailed analysis of an application (possibly during its development or porting to a new system), and also as input to resources allocation (scheduling) systems on-the-fly (at run-time).

An overview of PACE is given in the following sections. Section 2 describes the main components of the PACE system. Section 3 details an underlying language used within PACE detailing the performance aspects of the applications / systems. An application may be automatically translated to the internal PACE language representation. Section 4 describes how performance predictions are obtained in PACE. Examples of using PACE performance models for off-line and on-the-fly analysis for scheduling applications on distributed resources is included in Section 5.

2 The PACE System

PACE (Performance Analysis and Characterisation Environment) [2] is a performance prediction and analysis toolset whose potential users include application programmers without a formal training in modeling and performance analysis. Currently, high performance applications based on message passing (using MPI or PVM) are supported. In principal any hardware platform that utilises this programming model can be analysed within PACE, and the technique has been applied to various workstation clusters, the SGI origin systems, and the CRAY T3E to

date. PACE allows the simultaneous utilisation of more than one platform, thus supporting heterogeneous systems in meta-computing environments.

There are several properties of PACE that enable it to be used throughout the development, and execution (run-time scheduling), of applications. These include:

Lifecycle Coverage – Performance analysis can be performed at any stage in the software lifecycle [3,4]. As code is refined, performance information is updated. In a distributed execution environment, timing information is available on-the-fly for determining which resources should be used.

Abstraction – Different forms of workload information need to be handled in conjunction with lifecycle coverage, where many levels of abstraction occur. These range from complexity type analysis, source code analysis, intermediate code (compile time) analysis, and timing information (at run- time).

Hierarchical – PACE encapsulate necessary performance information in a hierarchy. For instance an application performance description can be partitioned into constituent performance models. Similarly, a performance model for a system can consist of many component models.

Modularity – All performance models incorporated into the analysis should adhere to a strict modular structure so that a model can easily be replaced and re-used. This can be used to give comparative performance information, e.g. for a comparison of different system configurations in a meta-computing environment.

The main components of the PACE tool-set are shown in Fig. 1. A core component of PACE is the performance language, CHIP^{3}S (detailed in Section 3) that describes the performance aspects of an application and its parallelisation. Other parts of the PACE system include:

Object Editor – to assist in the creation and editing of individual performance objects. Pre-defined objects can be re-used through an object library system.

Source Code Analysis – enables source code to be analysed and translated into CHIP^{3}S. The translation performs a static analysis of the code, and dynamic constructs are resolved either by profiling or user specification.

Compiler – translates the performance scripts into C language code, linked to an evaluation library and specific hardware objects, resulting in a self-contained executable. The performance model remains parameterised in terms of system configurations (e.g. processor mapping) and application parameters (data sizes).

Hardware Configuration – allows the definition of a computing environment in terms of its constituent performance model components and configuration information. An underlying Hardware Modeling and Configuration Language (HMCL) is used.

Evaluation Engine –combines the workload information with component hardware models to produce time predictions. The output can be either overall execution time estimates, or trace information of the expected application behavior.

Performance Analysis – both ‘off-line’ and ‘on-the-fly’ analysis are possible. Off-line analysis allows user interaction and can provide insights into expected performance. On-the-fly analysis facilitates dynamic decision making at run-time, for example to determine which code to be executed on which available system.

There is very little restriction on how the component hardware models can be implemented within this environment, which allows flexibility in their design and

implementation. Support for their construction is currently under development in the form of an Application Programming Interface (API) that will allow access to the CHIP^{3}S performance workload information and the evaluation engine.

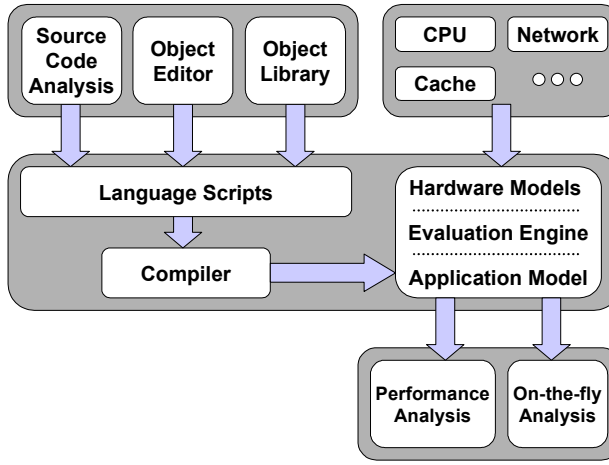


Fig. 1. Schematic of the PACE System.

3 Performance Language

A core component of PACE is the specialised performance language, CHIP^{3}S (Characterisation Instrumentation for Performance Prediction of Parallel Systems) [5] based on Software Performance Engineering principles [6]. This language has a strict object structure encapsulating the necessary performance information concerning each of the software and hardware components. A performance analysis using CHIP^{3}S comprises many objects linked together through the underlying language. It represents a novel contribution to performance prediction and evaluation studies.

3.1 Performance Object Hierarchy

Performance objects are organised into four categories: application, subtask, parallel template, and hardware. The aim of this organisation is the creation of independent objects that describe the computational parts of the application (within the application and subtask objects), the parallelisation strategy and mapping (parallel template object), and the system models (hardware object). The objects as follows:

Application Object - acts as the entry point to the performance model, and interfaces to the parameters in the model (e.g. to change the problem size). It also specifies the system being used, and the ordering of subtask objects.

Subtask Objects - represents one key stage in the application and contains: a description of sequential parts of the parallel program. These are modeled using CHIP^{3}S procedures which may be automatically formed from the source code.

Parallel Template Objects - describes the computation–communication pattern of a subtask object. Each contains steps representing a single stage of the parallel algorithm. A step defines the hardware resource.

Hardware Objects - The performance aspects of each system are encapsulated into separate hardware objects - a collection of system specification parameters (e.g. cache size, number of processors), micro-benchmark results (e.g. atomic language operations), statistical models (e.g. regression communication models), analytical models (e.g. cache, communication contention), and heuristics.

A hierarchical set of objects form a complete performance model. An example of a complete performance model, represented by a Hierarchical Layered Framework Diagram (HLFD) is shown in Fig. 2. The boxes represent the individual objects, and the arcs show the dependencies between objects in different layers.

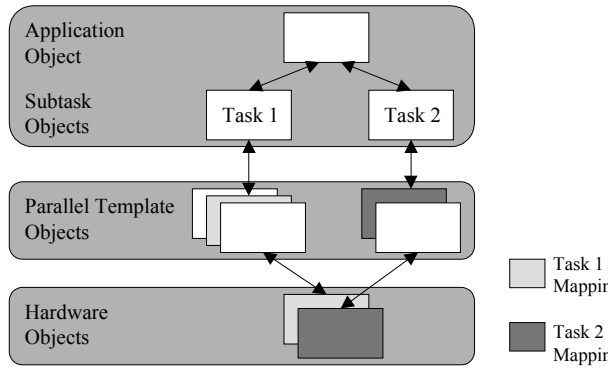


Fig. 2. Example HLFD illustrating possible parallelisation and hardware combinations.

In this example, the model contains two subtask objects, each with associated parallel templates and hardware objects. When several systems are available, there are choices to be made in how the application will be mapped. Such a situation is shown in Fig. 2 where there are three alternatives of mapping Task 1 (and two for Task 2) on two available systems. The shading also indicates the best mapping to these systems. Note that the two tasks are found to use different optimal hardware platforms.

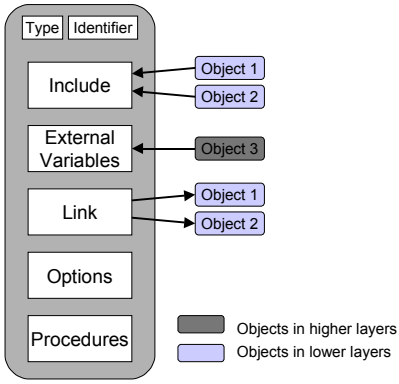


Fig. 3. Performance object structure

Include – references other objects used lower in the hierarchy.
External Variables – variables visible to objects above in the hierarchy.
Linking – modifies external variables of objects lower in the hierarchy
Option – sets default options for the object
Procedures –structural information for either: sub-task ordering (application object), computational components (sub-task objects), or computation / communication structure (parallel template objects).

3.2 Performance Object Definition

Each object describes the performance aspects of the corresponding system component but all have a similar structure. Each is comprised of internal structure (hidden from other objects), internal options (governing its default behavior), and an interface used by other objects to modify their behavior. A schematic representation of an object, in terms of its constituent parts, is shown in Fig. 3: A full definition of the CHIP^{3}S performance language is out of the scope of this paper [7].

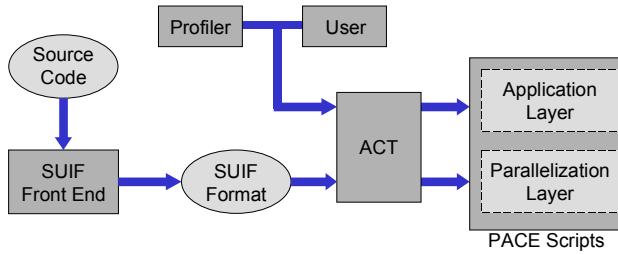


Fig. 4. Model creation process with ACT.

3.3 Software Objects

Objects representing software within the performance model are formed using ACT (*Application Characterisation Tool*). ACT provides semi-automated methods to produce performance models from existing sequential or parallel code for both the computational and parallel parts of the application, Fig. 4. Initially, the application code is processed by the SUIF front end [8] and translated into SUIF. The unknown parameters within the program (e.g. loop iterations, conditional probabilities) that cannot be resolved by static analysis are found either by profiling or user specification. ACT can describe resources at four levels:

Language Characterisation (HLLC) – using source code (C and Fortran supported).
Intermediate Format Code Characterisation (IFCC) - characterisation of compiler representation (SUIF, Stanford University Intermediate Format, is supported) .
Instruction Code Characterisation (ICC) – using host assembly.
Component Timings (CT) - application component benchmarked. This produces accurate results but is non-portable across target platforms.

3.4 Hardware Objects

For each hardware system modeled an object describes the time taken by each resource available. For example, this might be a model of the time taken by an inter-processor communication, or the time taken by a floating-point multiply instruction. These models can take many different forms, ranging, from micro-benchmark timings of individual operations (obtained from an available system) to complex analytical models of the devices involved. One of the goals of PACE is to allow hardware objects to be easily extended. To this end an API is being developed that will enable third party models to be developed and incorporated into the prediction system.

Hardware objects are flexible and can be expressed in many ways. Each model is described by an evaluation method (for the hardware resource), input configuration parameters, and access to relevant workload information. Three component models are included in Fig 5. The workload information is passed from objects in upper layers, and is used by the evaluation to give time predictions. The main benefit of this structure is the flexibility; analytical models may be expressed by using complex modeling algorithms and comparatively simple inputs, whereas models based on benchmark timings are easily expressed but have many input parameters.

To simplify the task of modeling many different hardware systems, a hierarchical database is used to store the configuration parameters associated with each hardware system. The Hardware Model Configuration Language (HMCL) allows users to define new hardware objects by specifying the system-dependent parameters. On evaluation, the relevant sets of parameters are retrieved, and supplied to the evaluation methods for each of the component models. In addition, there is no restriction that the hardware parameters need be static - they can be altered at run-time either to refine accuracy, or to reflect dynamically changing systems.

Component models currently in PACE include: computational models supporting HLLC, IFCC, ICC and CT workloads, communication models (MPI & PVM), and multi-level cache memory models [9]. These are all generic models (the same for all supported systems), but are parameterised in terms of specific system performances.

4 Model Evaluation

The evaluation engine uses the CHIP^{3}S performance objects to produce predictions for the system. The evaluation process is outlined in Fig. 5. Initially, the application and sub-task objects are evaluated, producing predictions for the workload. These predictions are then used when evaluating computation steps in the parallel templates.

Calls to each component hardware device are passed to the evaluation engine. A dispatcher distributes input from the workload descriptions to an event handler and then to the individual hardware models. The event handler constructs an event list for each processor being modeled. Although the events can be identified through the target of each step in the parallel template, the time spent using the device is still unknown at this point. However, each individual hardware model can produce a time prediction for an event based on its parameters. The resultant prediction is recorded in the event list. When all device requests have been handled, the evaluation engine processes the event list to produce an overall performance estimate for the execution time of the application (by examining all event lists to identify the step that ends last).

Processing the event list is a two-stage operation. The first stage constructs the events, and the second resolves ordering dependencies, taking into account contention factors. For example, in predicting the time for a communication, the traffic on the inter-connection network must be known to calculate channel contention. In addition, messages cannot be received until after they are sent! The exception to this type of evaluation is a computational event that involves a single CPU device - this can be predicted in the first stage of evaluation (interaction is not required with other events).

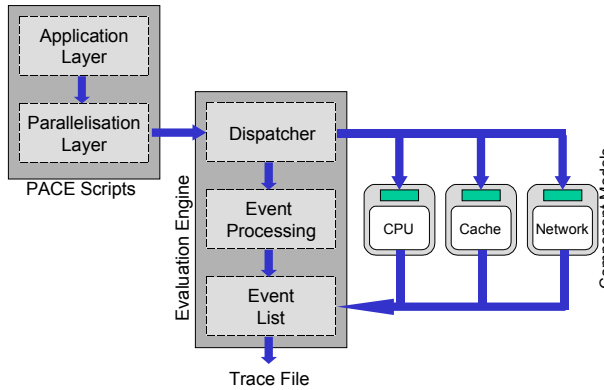


Fig 5. The evaluation process to produce a predictive trace within PACE.

The ability of PACE to produce predictive traces derives directly from the event list formed during model evaluation. Predictive traces are produced in standard trace formats. They are based on predictions and not run-time observations. Two formats are supported by PACE: PICL (Paragraph), and SDDF (PABLO) [10].

5 Performance Models in Use

The PACE system has been used to investigate many codes from several application domains including image processing, computational chemistry, radar, particle physics, and financial applications. PACE performance models are in the form of self-contained executable binaries parameterised in terms of application and system configuration parameters. The evaluation time of a PACE performance model is rapid

(typically seconds of CPU use) as a consequence of utilising many small analytical component hardware models. The rapid execution of the model lends itself to dynamic situations as well as traditional off-line analysis as described below.

5.1 Off-Line Analysis

A common area of interest in investigating performance is examining the execution time as system and/or problem size is varied. Application execution for a particular system configuration and set of problem parameters may be examined using trace analysis. Fig. 6 shows a predictive trace analysis session within Pablo. In the background, an analysis tree contains an input trace file (at its root node) and, using data manipulation nodes, results in a number of separate displays (leaves in the tree). Four types of displays are shown producing summary information on various aspects of the expected communication behavior. For example, the display in the lower left indicates the communication between source and destination nodes in the system (using contours to represent traffic), and the middle display shows the same information displayed using 'bubbles', with traffic represented by size and colour.

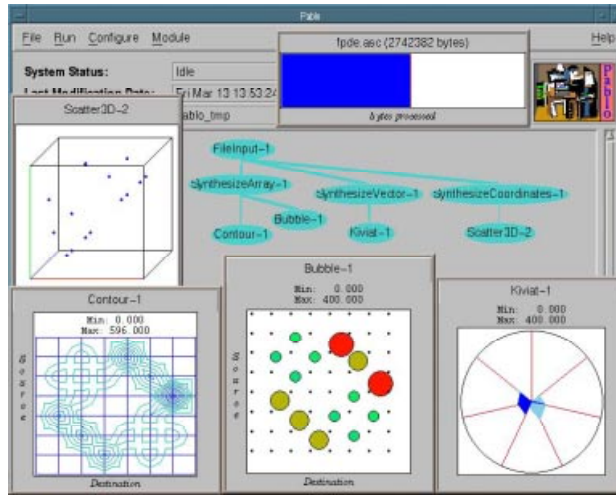


Fig. 6. Analysis of trace data in Pablo

5.2 On-the-Fly Analysis

An important application of prediction data is that of dynamic performance-steered optimisation [11,12] which can be applied for efficient system management. The PACE model is able to provide performance information for a given application on a given system within a couple of seconds. This enables the models to be applied on-the-fly for run-time optimisation. Thus, dynamic just-in-time decisions can be made about the execution of an application, or set of applications, on the available system

(or systems). This represents a radical departure from existing practice, where optimisation usually takes place only during the program's development stage

Two forms of on-the-fly analysis have been put into use by PACE. The first has involved a single image processing application, in which several choices were available during its execution [13]. The second is a scheduling system applied to a network of heterogeneous workstations. This is explained in more detail below.

The console window of the scheduling system, using performance information and a Genetic Algorithm (GA) is shown in Fig. 7. The coloured bars represent the mapping of applications to processors; the lengths of the bars indicate the predicted time for the number of processors allocated. The system works as follows:

1. (An application (and performance model) is submitted to the scheduling system.
2. The GA contains all the performance data for the currently submitted applications, and constantly minimises the execution time for the application set.
3. Applications currently executing are 'fixed' and cannot change the schedule.
4. Feedback updates the GA on premature completion, or late-running applications.

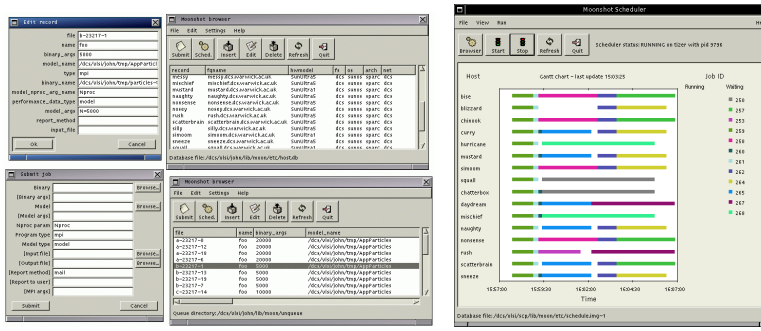


Fig. 7. Console screen of the PACE scheduling system showing the system and task interfaces (left panels) and a view of the Gantt chart of queued tasks (right panel).

One particular advantage of the GA method over the other heuristics tried is that it is an evolutionary process, and is therefore able to absorb slight changes, such as the addition or deletion of programs from its optimisation set, or changes in the resources available in the computing system.

6 Conclusion

This work has described a methodology for providing predictive performance information on parallel applications using a hierarchical model of the application, its parallelisation, and the distributed system. An implementation of this approach, the PACE toolset, has been developed over last three years, and has been used to explore performance issues in a number of application domains. The system is now in a

position to provide detailed information for use in static analysis, such as trace data, and on-the-fly analysis for use in scheduling and resource allocation schemes.

The speed with which the prediction information is calculated has led to investigations into its use in dynamic optimisation of individual programs and of the computing system as a whole. Examples have been presented of dynamic algorithm selection and system optimisation, which are both performed at run-time. These techniques have clear use for the management of dynamically changing systems and GRID based computing environments.

Acknowledgement

This work is funded in part by DARPA contract N66001-97-C-8530, awarded under the Performance Technology Initiative administered by NOSC.

References

1. I Foster, C Kesselman, „The GRID“, Morgan Kaufman (1998)
2. G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, D.V. Wilcox, „PACE – A Toolset for the Performance Prediction of Parallel and Distributed Systems“, in the *Journal of High Performance Applications*, Vol. 14, No. 3 (2000) 228-251
3. D.G. Green et al. „HPCN tools: a European perspective“, *IEEE Concurrency*, Vol. 5(3) (1997) 38-43
4. I. Gorton and I.E. Jelly, „Software engineering for parallel and distributed systems, challenges and opportunities“, *IEEE Concurrency*, Vol. 5(3) (1997) 12-15
5. E. Papaefstathiou et al., „An overview of the CHIP^{3}S performance prediction toolset for parallel systems“, in *Proc. of 8th ISCA Int. Conf. on Parallel and Distributed Computing Systems* (1995) 527-533
6. C.U. Smith, „Performance Engineering of Software Systems“, Addison Wesley (1990).
7. E. Papaefstathiou et al., „An introduction to the CHIP^{3}S language for characterising parallel systems in performance studies“, Research Report RR335, Dep. of Computer Science, University of Warwick (1997)
8. Stanford Compiler Group, *„*The SUIF Library*“*, *The SUIF compiler documentation set*, Stanford University (1994)
9. Harper, J.S., Kerbyson, D.J., Nudd, G.R.: Analytical Modeling of Set-Associative Cache Behavior, IEEE Transactions on Computers, Vol. 48(10) (1999) 1009-1024
10. D.A. Reed, et al., „Scalable Performance Analysis: The Pablo Analysis Environment“, in: *Proc. Scalable Parallel Libraries Conf.*, IEEE Computer Society (1993)
11. R. Wolski, „Dynamically Forecasting Network Performance Using the Network Weather Service“, UCSD Technical Report, TR-CS96-494 (1996)
12. J. Gehring, A. Reinefeld, „MARS - A framework for minizing the job execution time in a metacomputing environment“, *Future Generation Computer Systems*, Vol. 12 (1996) 87-99
13. D.J. Kerbyson, E. Papaefstathiou and G.R. Nudd, „Application execution steering using on-the-fly performance prediction“, in: *High Performance Computing and Networking*, Vol 1401, LNCS Springer-Verlag (1998) 718-727

Delay Behavior in Domain Decomposition Applications

Marco Dimas Gubitoso and Carlos Humes Jr.

Universidade de São Paulo
Instituto de Matemática e Estatística
R. Matão, 1010, CEP 05508-900 São Paulo, SP, Brazil
{gubi,humes}@ime.usp.br

Abstract. This paper addresses the problem of estimating the total execution time of a parallel program based on a domain decomposition strategy.
When the execution time of each processor may vary, the total execution time is non deterministic, specially if the communication to exchange boundary data is asynchronous.
We consider the situation where a single iteration on each processor can take two different execution times. We show that the total time depends on the topology of the interconnection network and provide a lower bound for the ring and the grid.
This analysis is supported further by a set of simulations and comparisons of specific cases.

1 Introduction

Domain decomposition is a common iterative method to solve a large class of partial differential equations numerically. In this method, the computation domain is partitioned in several smaller subdomains and the equation is solved separately on each subdomain, iteratively. At the end of each iteration, the boundary conditions of each subdomain are updated according to its neighbors.

The method is particularly interesting for parallel programs, since each subdomain can be computed in parallel by a separate processor with a high expected speedup. Roughly speaking, the amount of computation at each iteration is proportional to the volume (or area) of the subdomain while the communication required is proportional to the boundary.

A general form of a program based on domain decomposition is:

For each processor p_k, $\quad k = 1 \ldots P$:

```
DO I = 1, N
    < computation>
    < boundary data exchange >
END DO
```

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 160–167, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In this situation, a processor p_k must have all the boundary values available before proceeding with the next iteration. This forces a synchronization between p_k and its neighbors.

If the computation always takes the same time to complete, independent of the iteration, the total execution time has a simple expression. If T_{comp} and T_{exch} are the computation and communication (data exchange) times, respectively, the total parallel time, T_{par}, is given by:

$$T_{par} = N \cdot (T_{comp} + T_{exch})$$

In the sequential case, there is only one processor and no communication. The total time, considering N iterations and P sites, is then

$$T_{seq} = N \cdot P \cdot T_{comp}$$

In this simple case it does not matter if the communication is synchronous (i.e. with a blocking *send*) or asynchronous, since T_{comp} is the same for all processors. However, if a processor can have a random delay the type of communication has a great impact on the final time, as will be shown.

A sample situation where a delay can occur is when there is a conditional branch in the code:

```
DO I = 1, N
   IF(cond) THEN
     < computation 1 >
   ELSE
     < computation 2 >
   ENDIF
   < boundary data exchange >
ENDDO
```

In this paper, we will suppose the following hypothesis are valid:

1. The communication time is the same for all processors and does not vary from one iteration to another.
2. Any processor can have a delay δ in computation time with probability α. In the sample situation above, α is the true ratio of `cond` and *delta* is the difference in the execution times of $<$ computation 1 $>$ and $<$ computation 2 $>$, supposing the later takes less time to complete.
3. α is the same for all iterations.

If T_c is the execution time of a single iteration without a delay, the expected execution time for one iteration is $T_c + \alpha\delta$. In the sequential case, the total execution time is then

$$< T_{seq} >= N \cdot P \cdot (T_c + \alpha\delta)$$

and the distribution of probability is a binomial

$$\mathcal{P}(m \text{ delays}) = B(m, N, \alpha) = \binom{N}{m} \alpha^m (1-\alpha)^{N-m}$$

For the parallel case with synchronous communication the time of a single iteration is the time taken by the slowest processor. The probability of a global delay is the probability of a delay in at least one processor, that is (1 − probability of no delay)

$$\mathcal{P}(\text{delay in one iteration}) = \rho = 1 - (1-\alpha)^P \tag{1}$$

and the expect parallel execution is

$$< T_{par} >= N \cdot (T_{exch} + (T_c + \rho\delta))$$

It should be noticed that a processor can have a "spontaneous" or an "induced" delay. The delay is spontaneous if it is a consequence of the computation executed inside the processor. The processor can also suffer a delay while waiting data from a delayed neighbor.

1.1 Asynchronous Communication

If the communication is asynchronous, different processors may, at a given instant, be executing different instances of the iteration and the number of delays can be different for each processor. This is illustrated in the figure 1.

In this figure, each frame shows the number of delays on each processor in a 5×5 grid in successive iterations. At each instant, the processors which had a spontaneous delay are underlined. At the stage (f), even if four processors had a spontaneous delay, the total delay was three.

During data exchanging each processor must wait for its neighbors and the delays propagate, forming "bubbles" which expand until they cover all processors. This behavior can be modeled as follows:

1. Each processor p has a positive integer associated, $n_a[p]$, which indicates the total number of delays.
2. Initially $n_a[p] = 0, \forall p$.
3. At each iteration, for each p:
   - $n_a[p] \leftarrow \max\{n_a[i] | i \in \{\text{neighbors of } p\}\}$.
   - $n_a[p]$ is incremented with probability α.

The total execution time is given by

$$N \cdot (T_c + T_{exch}) + A \cdot \delta$$

where $A = \max\{n_a[p]\}$, after N iterations.

In the remaining of this paper, we will present a generic lower bound for A, a detailed analysis for the ring and the grid interconnection communication networks, a set of simulations and conclude with qualitative analysis of the results.

0 0 0 0 0	0 0 0 0 0	0 1 0 0 0
0 0 0 0 0	0 $\underline{1}$ 0 0 0	1 1 1 0 0
0 0 0 0 0	0 0 0 0 0	0 1 0 0 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 $\underline{1}$
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
(a)	(b)	(c)

1 1 1 0 0	1 1 1 1 0	2 1 1 1 1
1 1 1 1 0	2 1 1 1 1	2 2 1 1 1
$\underline{2}$ 1 1 0 1	2 2 1 1 1	2 $\underline{3}$ 2 1 1
0 1 0 1 1	2 1 1 1 1	2 2 1 1 1
0 0 0 0 1	0 1 0 1 1	2 2 1 1 1
(d)	(e)	(f)

Fig. 1. Delay propagation in the processors

2 Lower Bound for the Number of Total Delays

The probability of a delay in any given processor is a combination of the induced and spontaneous possibilities. Let γ be this joint probability. It is clear that $\gamma \geq \alpha$, for α corresponds only to the spontaneous delay.

An increase in the total delay only happens if one of the most delayed processors suffers a new (spontaneous) delay.

The number of delays a processor has is called "level".

To find a lower bound, we choose one processor among the most delayed and ignore the others. In other words, we choose a single bubble and do not consider the influence of others. We then compute the expected delay in l successive iterations, to find an approximation for γ. We call this procedure a 'l-look-ahead' estimate.

A spontaneous delay can only happen on the computation phase of an iteration. At this time, any processor inside the bubble can have its level incremented (with probability α).

In the communication phase, the bubble grows[1] and the level of a processor can be increased by one of its neighbors. On the approximation bubble we are considering, only processors inside the bubble can change the level of a processor.

$$0 \Rightarrow 0 \rightarrow 000 \Rightarrow 100 \rightarrow 11100 \Rightarrow 11200 \rightarrow 1122200$$
$$0 \Rightarrow 1 \rightarrow 111 \Rightarrow 112 \rightarrow 11222 \Rightarrow 11232 \rightarrow 1123332$$

Fig. 2. Some possible single bubble evolutions for a ring

[1] Unless it touches another bubble at a higher level, but we are considering the single bubble case.

Figure 2 illustrates a sample evolution of a single bubble on a 1-dimensional network. Each transformation indicated by $\rightarrow$ is a propagation expanding the size of the bubble. This expansion depends on the topology. The $\Rightarrow$ transformation is related to spontaneous delays and depends only on the current bubble size.

2.1 Transition Probability

In order to derive the expression for γ, we state some definitions and establish a notation:

- A bubble is characterized by an array of levels, called *state*, indicated by S, and a propagation function that depends on the topology of the interconnection network.
- The state represents the delay of each processor inside the bubble: $S = s_1 s_2 \cdots s_k \cdots s_C$, where $C = |S|$ is the bubble size. s_k is the delay (level) of the kth processor in the bubble.
- $s_{max} = \max\{s_1, \ldots, s_C\}$
- $S' = s'_1 \cdots s'_{C'}$ is an expansion of S, obtained by a propagation.
- $n(S)$ is the number of iterations corresponding to S, that is, the time necessary to reach S. For the ring: $|S| = 2 \cdot n(S) + 1$
- The *weight* of a level in a bubble S is defined as follows:

$$W_S(k) = \begin{cases} \text{number of occurrences of } k \text{ in } S, & \text{if } k = \max\{s_1, \ldots, s_C\} \\ 0, & \text{otherwise} \end{cases}$$

- The *set of generators* of a bubble S, $G(S)$ is the set of states at $t = n(S) - 1$ which can generate S by spontaneous delays.
  If $T \in G(S)$ then$|T'| = |S|$, $n(T) = n(S) - 1$ and $P(T \text{ reach } n(S)) = 1 - (1-\alpha)^{W_{T'}(n(T))}$
- The number of differences between two bubbles S and T is indicated by $\Delta(S,T)$:

$$\Delta(S,T) = \sum_{i=1}^{C} (1 - \delta_{s_i t_i})$$

  where $\delta_{s_i t_i}$ is the Kronecker's delta.
- A *chain* is a sequence of states representing a possible history of a state S:

$$\mathcal{C}(S) = S^0 \rightarrow S^1 \rightarrow \cdots \rightarrow S^a, \quad S^a \in G(S)$$

Consider S^i and S^{i+1} two consecutive states belonging to the same chain. For S^i to reach S^{i+1}, processors with different levels in S'^i andS^{i+1} must suffer spontaneous delays. All the other processors cannot change their state. The transition probability is then:

$$P(S^i \rightarrow S^{i+1}) = \alpha^{\Delta(S'^i, S^{i+1})} \cdot (1-\alpha)^{|S'^i| - \Delta(S'^i, S^{i+1})}$$

and the probability if a specific chain $\mathcal{C}(S)$ to occur is given by:

$$\mathcal{P}(\mathcal{C}(S)) = \prod_{i=0}^{a-1} \alpha^{\Delta(S'^i, S^{i+1})} \cdot (1-\alpha)^{|S'^i| - \Delta(S'^i, S^{i+1})}$$

2.2 Effective Delay

The total delay associated with a state S is s_{max}. If S is the final state, for a given $S^a \in G(S)$, the final delay can be:

1. s^a_{max} with probability $(1-\alpha)^{W_{s^a}(s^a_{max})}$, that is, when none of the most delayed processors has a spontaneous delay, or
2. $s^a_{max}+1$, if at least one of these processors has a new delay. The probability for this to happen is $1-(1-\alpha)^{W_{s^a}(s^a_{max})}$

The expected delay E is computed over all possible histories:

$$E = \sum_{\mathcal{C}(S)} \mathcal{P}(\mathcal{C}(S)) \cdot \Big(s^a_{max} \cdot (1-\alpha)^{W_{s^a}(s^a_{max})} + (s^a_{max}+1) \cdot \Big[1-(1-\alpha)^{W_{s^a}(s^a_{max})}\Big]\Big) \tag{2}$$

$$E = \sum_{\mathcal{C}(S)} \mathcal{P}(\mathcal{C}(S)) \cdot \Big((s^a_{max}+1) - (1-\alpha)^{W_{s^a}(s^a_{max})}\Big) \tag{3}$$

and the effective delay, γ, for the l-look-ahead is E/l.

For any regular graph of degree d, the 1-look-ahead approximation has a simple expression for γ. In any history there are only two states, S_0 and S_1, with $|S_0| = 1$ and $|S_1| = d+1$, and the probability of a new delay is (remember that $l = 1$):

$$\gamma = \big(1-(1-\alpha)^{d+1}\big) \tag{4}$$

Equation 4 indicates that a larger degree implies a greater delay. In particular, a grid may provide a larger execution time than a ring with the same number of processors. The reason is that in a highly connected graph the bubbles expand more quickly.

For larger values of l, equation 3 becomes more complicated as the number of terms grows exponentially. Even so, it was not difficult to write a perl script to generate all the terms for 2 and 3 look-ahead for the ring and the grid and feed the results into a symbolic processor (Maple) to analyze the expression and plot a curve showing the effective delay as a function of α. These results are presented in the next section, together with the simulations.

3 Simulations

In this section, we present and discuss the simulation results for several sizes of grids and rings.

Each simulation had 500 iterations and the number of delays averaged over 20 executions, for α running from 0.0 to 1.0 by steps of 0.05.

The figures 3(a) and 3(b) show the effective delay ($\frac{A}{N}$) for a ring of 10 processors and a 5×5 grid, respectively. The bars indicate the minimum and the maximum delay of the 20 executions.

Figure 4 presents the comparison for 2 topologies with the same number of processors: 30×30 grid and a ring with 900 processors. It indicates that, for certain values of α and δ, the ring can be much better if the communication cost is not much higher (in a ring the boundary size of each subdomain is larger). For instance, for α between 0.10 and 0.30, the difference in the effective delay is over 20%.

A series of tests was made on a Parsytec PowerXplorer using 8 processor in a ring topology. The program used was a simple loop with neighbors communication and the computation at each iteration can include a large delay, following the model presented here, details of the implementation and further results are available upon request. Two representative curves (`Run1` and `Run2`) are shown in figure 5.

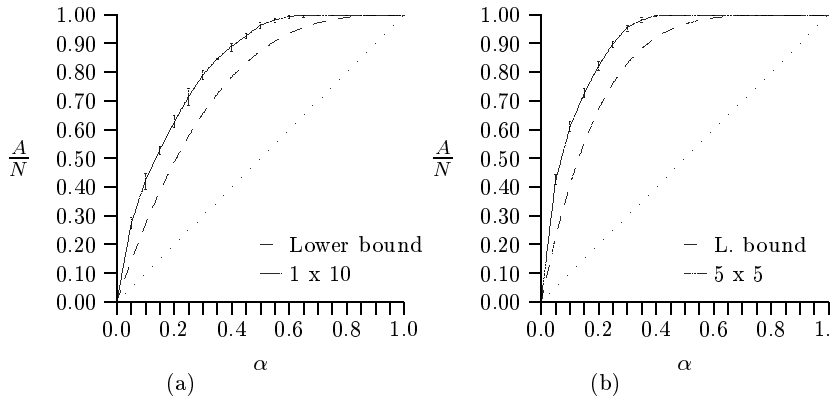


Fig. 3. Effective delay for (a) an array of 10 processors and (b) a grid of size 5

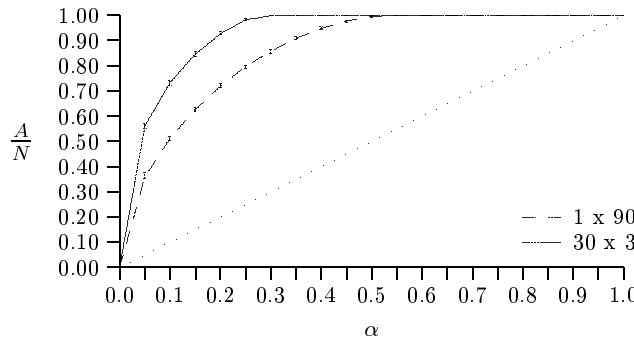


Fig. 4. Effective delay for a 30×30 grid

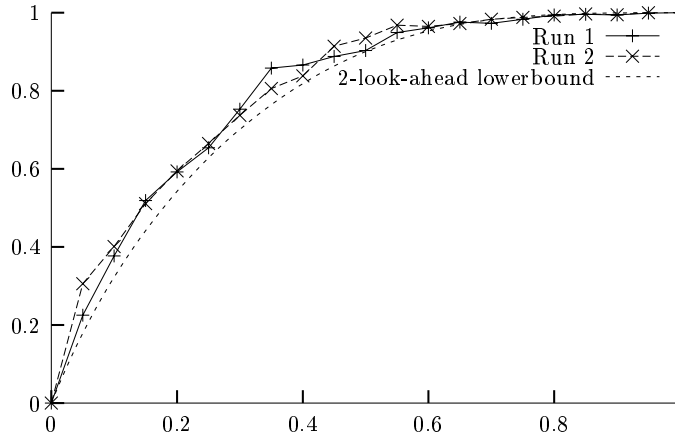


Fig. 5. Experiments in a real machine

4 Conclusions

We presented a method to derive a lower bound total delay for a domain decomposition application with a random delay at each iteration, under asynchronous communication. Inspection indicates very little difference between the 2 and 3 look-ahead approximations. Experiments in a real machine indicate that our lower bound expression is closer to the actual time observed than those obtained by simulation.

The expressions obtained and the simulations indicate that under *some* situations it is better to use a ring topology instead of a higher connected graph.

References

[1] Tony F. Chan. Parallel complexity of domain decomposition methods and optimal coarse grid sizes. *Parallel Computing*, To appear, 1994.

[2] William D. Gropp. Parallel computing and domain decomposition. In David E. Keyes, Tony F. Chan, Gérard A. Meurant, Jeffrey S. Scroggs, and Robert G. Voigt, editors, *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 349–361, Philadelphia, PA, 1992. SIAM.

[3] Marco Dimas Gubitoso and Carlos Humes jr. Delays in asynchronous domain decomposition. *Lecture Notes in Computer Science*, Maro 1999.

[4] Leslie Hart and Steve McCormick. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Computational analysis. *Parallel Computing*, 12:131-144, 1989.

Automating Performance Analysis from UML Design Patterns

Omer F. Rana[1] and Dave Jennings[2]

[1] Department of Computer Science, University of Wales, Cardiff, POBox 916, Cardiff CF24 3XF, UK
[2] Department of Engineering, University of Wales, Cardiff, POBox 916, Cardiff CF24 3XF, UK

Abstract. We describe a technique for deriving performance models from design patterns expressed in the Unified Modelling Language (UML) notation. Each design pattern captures a theme within the `Aglets` mobile agent library. Our objective is to find a middle ground between developing performance models directly from UML, and to deal with a more constrained case based on the use of design patterns.

1 Introduction

UML has become a widely talked about notation for object oriented design, and it is therefore often felt that the adoption of UML by the software engineering community makes it a good candidate for which performance evaluation techniques should be developed. Although in agreement with the approach, we feel that UML is too complex at present. Deciding the particular idiom to use for a given problem is also not clear when using UML, and it is often difficult to express concurrent and non-deterministic behaviour using it.

Design patterns capture recurring themes in a particular domain, facilitating re-use and validation of software components. Design patterns fill the gap between high level programming languages and system level design approaches. Our approach avoids proprietary extensions to UML, and makes use of well defined Petri net blocks, which model design patterns with a given intent, and in a particular context. We suggest that this is a more tractable approach, can can facilitate approximate analysis of larger applications via compositionality of design patterns, compared to other similar work [2, 3, 4, 6]. We illustrate our approach with the 'Meeting Design Pattern' in the Aglets workbench, from [1].

2 The Meeting Design Patterns

Intent: The Meeting pattern provides a way for agents to establish local interactions on specific hosts **Motivation:** Agents at different sites may need to interact locally at any given site. A key problem is the need to synchronise the operations performed by these agents, initially created at different hosts, and then dispatched to a host to find each other and interact. The Meeting pattern

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 168–172, 2000.
© Springer-Verlag Berlin Heidelberg 2000

helps achieve this, using a `Meeting` object, that encapsulates a meeting place (a particular destination), and a unique meeting identifier. An agent migrates to a remote host, and uses the unique meeting identifier to register with a meeting manager. Multiple meetings can be active on a given host, with a unique identifier for each meeting. The meeting identifier is described by an ATP address, such as `atp://emerald.cs.cf.ac.uk/serv1:4344`. The meeting manager then notifies already registered agents about the new arrival and vice versa. On leaving the meeting place, an agent de-registers via the meeting manager.
Applicability: The pattern is applicable when, (1) communication with a remote host is to be avoided, whereby the agent encapsulates business logic, and carries this to the remote machine, (2) where security or reliability constraints prevent interaction between software directly, such as across firewalls, (3) when local services need to be accessed on a given host.
Participants: There are three participants involved in the meeting, (1) a `Meeting` object that stores the address of the meeting place, a unique identifier, and various other information related to the meeting host, (2) a `Meeting Manager` which registers and de-registers incoming agents, announces arrival of new agents to existing ones etc, (3) an `Agent` base class, and an associated sub-class `ConcreteAgent`, from which each mobile agent is derived, and which maintains the meeting objects respectively.
Collaboration: A `Meeting` object is first created, an agent is then dispatched to the meeting place, where it notifies the `Meeting Manager`, it is then registered by the `addAgent()` method. On registration, the newly arrived agent is notified via the `meet()` method of all agents already present, and existing agents are notified via the `meetWith()` method. When an agent leaves, the `deleteAgent()` method is invoked.
Consequences: The meeting pattern has both advantages and drawbacks:

- Advantages: supports the existence of dynamic references to agents, and therefore facilitates the creation of mobile applications. It also enables an agent to interact with an unlimited number of agents at a given site, which can support different services (identified by different meeting managers). The meeting manager can act as an intermediary (called a Mediator), to establish multicast groups at a given host, enabling many-to-many interactions.
- Disadvantages: Some agents may be idle, waiting for their counterparts to arrive from a remote site.

3 Petri Net Models

To extract performance models from design patterns, we consider two aspects: (1) participants within the pattern, (2) collaboration between participants, and specialised constraints that are required to manage this collaboration. The participants and constraints help identify control places within Petri nets (labeled with circles containing squares), and collaboration between participants helps decide the types of transitions required – timed, intermediate or stochastic. Since a sequence diagram only expresses one possible interaction – a scenario, we do not

directly model a sequence diagram, but use it to determine message transfers between agents. Times associated with transitions can either be deterministic only, and based on simulation of the design pattern, or a designer can associate 'likely' times, to study the effect on overall system performance. More details about Petri net terminology can be found in [5].

We model two scenarios in our Petri net model: the first involves only a single agent service at a host, suggesting that all incoming agents are treated identically, and handled in the same way. In the second scenario, we consider multiple agent services active at a host, requiring that we first identify the particular agent service of interest. Hence, three Petri nets are derived: (1) Petri net for agent *arrival* and *departure*, (2) Petri net for *agent-host* interaction, (3) Petri net for *agent-agent* interaction.

3.1 Arrival/Departure Petri Nets

When a single agent service is present at a host, all incoming agents are treated identically. Each agent header is examined to determine the service it requires, and needs to be buffered at a port before passing on to the *MeetingManager* for registration. For multiple agent services, we use a colour to associate an incoming agent with a particular agent service.

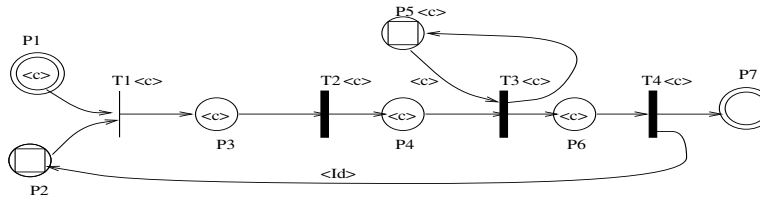


Fig. 1. *Arrival Petri net*

The Petri net in figure 1 identifies the receiving and registering of an incoming agent. We assume that only one *MeetingManager* is available, and consequently, only one agent can be registered at a time. The *MeetingManager* therefore represents a synchronisation point for incoming agents. From figure 1: Places *P1* and *P7* represent the start s_o and s_e place respectively. A token in place *P1* represents the existence of an incoming agent, and a token in *P2* models the presence of a particular port on which the agent is received. Place *P3* represents the buffering associated with the incoming agent at a port. Place *P4* is used to model the registration of the agent with the local *MeetingManager*, the latter being modelled as place *P5*. Place *P6* corresponds to the *notification* to existing agents of the presence of the new agent. Place *P7* is then used to trigger the next Petri net block which models either an *agent-agent* or *agent-host* interaction. Transition *T1* is an immediate transition, and models the synchronisation between an incoming agent and the availability of a particular port and buffer.

T1 would block if the port is busy. Transition *T2* is a timed transition, and represents the time to buffer the incoming agent. Transition *T3* represents the time to register and authenticate an incoming agent via the *MeetingManager*. We assume that only one agent can be registered at a time, hence the synchronisation of places *P5* and *P4* via *T3*. Marking $\mathcal{M}_0(P2)$ equals the number of connections accepted on the port on which the agent service listens for incoming requests and $\mathcal{M}_0(P5)$ is always equal to 1. The initial marking $\mathcal{M}_0$ can be specified as: $\mathcal{M}_0 = \{0, connections, 0, 0, 1, 0, 0\}$. Other PN models can be found in [7]. To model an agent system we combine the Petri net blocks above to capture particular agent behaviour as specified in the *Aglets* source code. Hence, we combine an *arrival* Petri net with an *agent-agent* or an *agent-host* Petri net, depending on the requested operations. The use of start and end places allows us to cascade Petri net blocks, with the end place of an *arrival* Petri net feeding into the start place of an *agent-agent* or *agent-host* Petri net. The end place of an *agent-agent* or an *agent-host* Petri net is then fed back to the start place of either an *agent-agent* or an *agent-host* Petri net if a repeated invocation is sought. Alternatively, the end place of an *agent-host* or an *agent-agent* Petri net indicates the departure of an agent from the system. In theory the model can be scaled to as many agents as necessary, being restricted by the platform on which simulation is performed. We have tested the model to 50 agents on 10 hosts.

4 Conclusion

We describe a general framework for building and managing mobile agent systems, based on the existence of agent design patterns in UML. A design pattern is modelled as a self contained Petri net block, that may be cascaded. Our model accounts for differences in the size of an MA, different agent services at a host and stochastic parameters such as the time to register an agent and port contention at a host. The Petri net provides us with a mathematical model, that may be analysed to infer properties of the mobile agent system.

References

[1] Y. Aridor and D. Lange. Agent Design Patterns: Elements of Agent Application Design. *Second International Conference on Autonomous Agents (Agents '98), Minneapolis/St. Paul - May 10-13*, 1998.

[2] T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a Semantics and its Application in the development process. *Technical Report 11/98, Institut für Informatik, Universität Hildesheim, Germany*, 1998.

[3] H. Giese, J. Graf, and G. Wirtz. Closing the Gap Between Object-Oriented Modeling of Structure and Behavior. *Proceedings of the Second International Conference on UML, Fort Collins, Colorado, USA*, October 1999.

[4] P. Kosiuczenko and M. Wirsing. Towards an integration of message sequence charts and timed maude. *Proceedings of the Third International Conference on Integrated Design and Process Technology, Berlin*, July 1998.

[5] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, April 1989.
[6] H. Störrle. A Petri-net semantics for Sequence Diagrams. *Technical Report, Ludwig-Maximilians-Universität München, Oettingenstr. 67, 80538 München, Germany*, April 1999.
[7] Omer F. Rana and Chiara Biancheri. A Petri Net Model of the Meeting Design Pattern for Mobile-Stationary Agent Interaction, *HICSS32*, January 1999

Integrating Automatic Techniques in a Performance Analysis Session

Antonio Espinosa, Tomas Margalef, Emilio Luque

Computer Science Department
Universitat Autonoma of Barcelona
08193 Bellaterra, Barcelona, SPA1N
{ antonio.espinosa, tomas.margalef, emilio.luque }@uab.es

Abstract. In this paper, we describe the use of an automatic performance analysis tool for describing the behaviour of a parallel application. KappaPi tool includes a list of techniques that may help the non-expert users in finding the most important performance problems of their applications. As an example, the tool is used to tune the performance of a parallel simulation of a forest fire propagation model

1. Introduction

The main reason for designing and implementing a parallel application is to benefit from the resources of a parallel system[1]. That is to say, one of the main objectives is to get a satisfing level of performance from the application execution. The hard task of building up an application using libraries like PVM[2], MPI[3] must yield the return of a fast execution in some cases or a good scalability in others, if not a combination of both.

These requirements usually imply a final stage of performance analysis. Once the application is running correctly it is time to analyse whether it is getting all the power from the parallel system it is running on.

To get a representative value of the level of performance quality of an application is necessary to attend to many different sources of information. They range from the abstract summary values like accumulated execution or communication time, to the specific behaviour of certain primitives. Middle solutions are available using some visualization tools [4,5].

The main problem with the performance analysis is the enormous effort that is required to understand and improve the execution of a parallel program. General summary values can focus the analysis in some aspects leaving others not so apparently crucial. For example, the analysis can be focused on communication aspects when the average waiting times are high. But then, the real analysis begins.

It seems rather manageable to discover the performance flaws of a parallel application from this general sources of information. Nevertheless, there is a considerable step to take to really know the causes of the low performance values detected. A great deal of information is required at this time, like which are the

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 173-177, 2000.
© Springer-Verlag Berlin Heidelberg 2000

processes that are creating the delay, what operations are they currently involved with, or which is the desired behaviour of the primitives used by the processes, and in what differs from the actually detected execution behaviour.

In other words, it is necessary to become an expert in the knowledge of the behaviour of the parallel language used and its consequences in the general execution.

To avoid this difficulty of becoming an expert in the performance analysis, a second generation of performance analysers have been developed. Tools like Paradyn [6], AlMS[7], Carnival[8] and P3T[9] have helped the users in this effort of the performance analysis introducing some automatic techniques that alleviate the difficulty of the analysis.

In this paper we present the use of Kappa Pi tool [10] for the automatic analysis of message-passing parallel applications. Its purpose is to deliver some hints about the performance of the application concerning the most important problems found and a possible suggestion about what can be done to improve the behaviour.

2. KappaPi Tool. Rule-Based Performance Analysis System

KappaPi is an automatic performance analysis tool for message-passing programs designed to provide some explanations to the user about the performance quality achived by the parallel program in a certain execution.

KappaPi takes a trace file as input together with the source code file of the main process (or processes) of the application. From there on, the objective of the analysis is to find the most important performance problems of the application looking at the processors' efficiency values expressed at the trace file events. Then, it tries to find any relationship between the behaviour found and any source code detail that may reveal information about the structure of the application.

The objective of KappaPi then is to help the programmers of applications to understand the behaviour of the application in execution and how it can be tuned to obtain the maximum efficiency.

First of all, the events are collected and analysed in order to build a summary of the efficiency along the execution interval studied. This summary is based on the simple accumulation of processor utilization versus idle and communication time. The tool keeps a table with those execution intervals with the lowest efficiency values.

At the end of this initial analysis we have an efficiency index for the application that gives an idea of the quality of the execution. On the other hand, we also have a final table of low efficiency intervals that allows us to start analyzing why the application does not reach better performance values.

The next stage in the KappaPi analysis is the classification of the most important inefficiencies. KappaPi tool identifies the selected inefficiency intervals with the use of a rule-based knowledge system. It takes the corresponding trace events as input and applies a set of behaviour rules deducing a new list of facts. These rules will be applied to the just deduced facts until the rules do not deduce any new fact. The higher order facts (deduced at the end of the process) allow the creation of an explanation of the behaviour found to the user. These higher order facts usually

represent the abstract structures used by the programmer in order to provide hints related to programming structures rather than low level events like communications.

The creation of this description depends very much on the nature of the problem found, but in the majority of cases there is a need of collecting more specific information to complete the analysis. In some cases, it is necessary to access the source code of the application and to look for specific primitive sequence or data reference. Therefore, the last stage of the analysis is to call some of this "quick parsers" that look for very specific source information to complete the performance analysis description.

3. Examining an Application: Forest Fire Propagation

The Forest Fire Propagation application (Xfire)[11] is a PVM message passing application that follows a master-worker paradigm where there is a single master process which generates the partition of the fireline and distributes it to the workers. These workers are in charge of the local propagation of the fire itself and have to communicate the position of the recently calculated fireline limits back to the master. The next step is to apply the general model with the new local propagation results to produce a new partition of the general fireline.

The first trace segment is analysed looking for idle or blocking intervals at the execution. A typical idle interval is the time waiting for a message to arrive when calling a blocking receive. All these intervals are identified by the id's of the processes involved and a label that describes the primitive that caused the waiting time.

Once the blocking intervals have been found, Kpi will use a rule-based system to classify the inefficiencies using a deduction process. The deduced facts express, at the beginning, a rather low level of expression like "communication between master and worker1 in machine 1 at lines 134, 30", which is deduced from the send-receive event pairs at the trace file. From that kind of facts some others are built on, like "dependency of worker1 from master" that reflects the detection of a communication and a blocking receive at process fireslave. From there, higher order level facts are deduced appling the list of rules. In the case of Xfire application, Kpi finds a master/worker collaboration. Once this collaboration has been detected, with the help of the ruled-based system, Kpi focuses on the performance details of such collaboration.

The first situation found when analysing Xfire in detail, is that all processes classified as master or worker in the deduced facts wait blocked for similar amounts of time, being the master the one that slightly accumulates more waiting time. This seems to mean that there is no much overlapping between the generation and the consumption of data messages.

On one hand, while the master generates the data, the worker waits in the reception of the next data to process. On the other hand, while the workers calculate the local propagation of the positions received, the master waits blocked to get back the new positions of the fireline boundaries. Then, the reception of messages at the master provokes a new generation of data to distribute.

Once this kind of collaboration is found, KappaPi tool tries to find which is the configuration of master-workers that could maximize the performance of the application. In this case, the key value is the adequate number of slaves to minimize the waiting times at the master.

To build a suggestion to the programmer, Kpi estimates the load of the calculation assigned to each worker (assuming that they all receive a similar amount of work). From there, Kpi calculates the possible benefits of adding new workers (considering the target processor's speed and communication latencies). This process will end when Kpi finds a maximum estimated number of workers to reduce the waiting times.

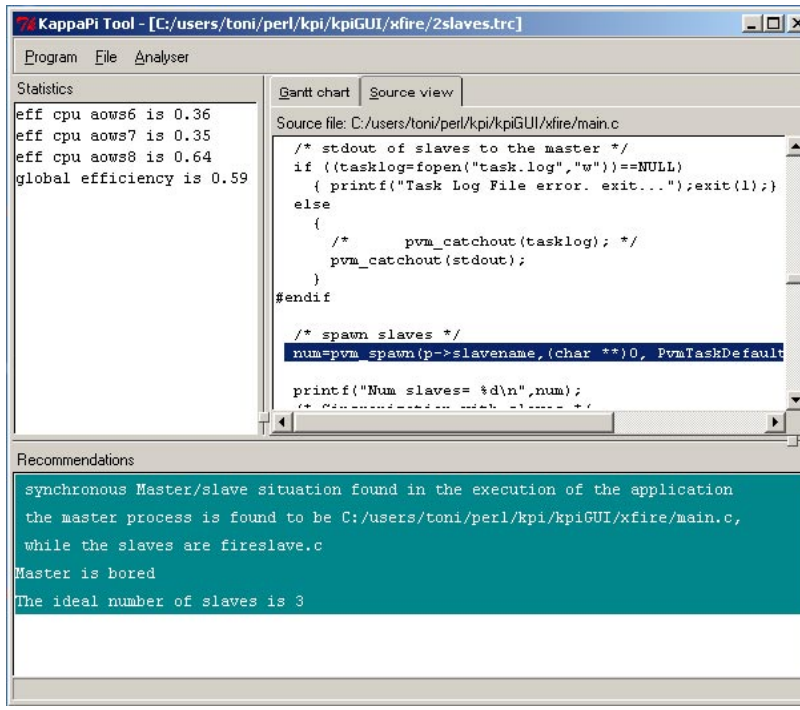


Fig. 1. Final view of the analysis of the Xfire application

Figure 1, shows the feedback given to the users of Kpi when the performance analysis is finished. The program window is split in three main areas, on the left hand side of the screen [statistics] there is a general list of efficiency values per processor. On the bottom of the screen [recommendations] the user can read the performance suggestion given by Kpi. On the right hand side of the screen [source view], the user can switch between a graphical representation of the execution (Gantt chart) and a view of the source code, with some highlighted critical lines that could be modified to improve the performance of the application. In the recommendations screen, the tool suggests to modify the number of workers in the application suggesting three as the best number of workers. Therefore, it points at the source code line where the spawn of the workers is done. This is the place to create a new worker for the application.

4. Conclusions

In conclusion, Kpi is capable of automatically detect a high level programming structure from a general PVM application with the use of its rule-based system. Furthermore, the performance of such an application will be analysed with the objective of finding which are their limits in the running machine. This process has been shown using a forest fire propagation simulator.

5. Acknowledgments

This work has been supported by the CYCIT under contract TIC98-0433

References

[1] Pancake, C. M., Simmons, M. L., Yan J. C.: „Perfonnance Evaluation Tools for Parallel and Distributed Systems“. lEEE Computer, November 1995, vol. 28, p. 16-19.

[2] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., „PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing“. MIT Press, Cambridge, MA, 1994.

[3] Gropp W., Nitzberg B., Lusk E., Snir M.: „Mpi: The Complete Reference: The Mpi Core/the Mpi Extensions. Scientific and Engineering Computation Series“. The MIT Press. Cambridge, MA, 1998.

[4] Heath, M. T., Etheridge, J. A.: „Visualizing the performance of parallel programs“. IEEE Computer, November 1995, *vol.* 28, p. 21-28.

[5] Reed, D. A., Giles, R. C., Catlett, C. E.. „Distributed Data and Immersive Collabolation“. Communications of the ACM. November 1997. Vol. 40, No 11. p. 3948.

[6] Hollingsworth, J. K., Miller, B, P.: „Dynamic Control of Performance Monitoring on Large Scale Parallel Systems“. International Conference on Supercomputing (Tokyo, July 1993).

[7] Yan, Y. C., Sarukhai, S. R.: „Analyzing palallel program performance using normalized performance indices and trace transformation techniques“. Parallel Computing 22 (1996) 1215-1237.

[8] Crovella, M.E. and LeBlanc, T. J.: „The search for Lost Cycles: A New approach to parallel performance evaluation“. TR479. The Unhersity of Rochester, Computer Science Department, Rochester, New York, December 1994.

[9] Fahringer T.: „Automatic Performance Prediction of Parallel Programs“. Kluwer Academic Publishers. 1996.

[10] Espinosa, A., Margalef, T. and Luque, E.: „Automatic Performance Evaluation of Parallel Programs“. Proc. of the 6th EUROMICRO Workshop on Parallel and Distributed Processing, pp. 4349. IEEE CS. 1998. http://www.caos.uab.es/kpi.html

[11]Jorba, J., Margalef, T., Luque, E., Andre, J., Viegas, D. X.: "Application of Parallel Computing to the Simulation of Forest Fire Propagation". Proc. 3td International Conference in Forest Fire Propagation, Vol. 1, pp. 891-900, Luso, Nov. 1998.

Combining Light Static Code Annotation and Instruction-Set Emulation for Flexible and Efficient On-the-Fly Simulation

Thierry Lafage and André Seznec

IRISA, campus de Beaulieu, 35042 Rennes cedex, France
{Thierry.Lafage, Andre.Seznec}@irisa.fr

Abstract This paper proposes a new cost effective approach for on-the-fly microarchitecture simulations on real size applications. The original program code is lightly annotated to provide a fast (direct) execution mode, and an embedded instruction-set emulator enables on-the-fly simulations. While running, the instrumented-and-emulated program can switch from the fast mode to the emulation mode, and vice-versa. The instrumentation tool, **calvin2**, and the instruction-set emulator, DICE, presented in this paper, exhibit low execution overheads in fast mode (1.31 on average for the SPEC95 benchmarks). This makes our approach well suited to simulate on-the-fly samples spread over an application.

1 Introduction

Simulations are widely used to evaluate microprocessor architecture and memory system performance. Such simulations require dynamic information (trace) of realistic programs to provide realistic results. However, compared to the native execution of the programs, microarchitecture (or memory system) simulation induces very high execution slowdowns (in the 1,000–10,000 range [1]).

To reduce simulation times, trace sampling, as suggested by [5] is widely used. For long running workloads, using the complete trace to extract samples is not conceivable because storing the full trace would need hundreds of giga bytes of disk, and would take days: on-the-fly simulation is the only acceptable solution. However, current trace-collection tools are not really suited to such a technique: at best, they provide a "skip mode" (to position the simulation at a starting point) which still exhibits a high execution overhead. Thus, using current tracing tools, trace sampling does not allow to simulate on-the-fly samples spread over long applications.

This paper presents the implementation of a new on-the-fly simulation approach. This approach takes advantage of both static code annotation and instruction-set emulation in order to provide traced programs with two execution modes: a fast (direct) execution mode and an emulation mode. At run time, dynamic switches are allowed from fast mode to emulation mode, and vice-versa. The fast execution mode is expected to exhibit a very low execution overhead over the native program execution, and therefore will enable to fast forward billions of

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 178–182, 2000.
© Springer-Verlag Berlin Heidelberg 2000

instructions in a few seconds. In addition, the instruction-set emulator is flexible enough to allow users to easily implement different microarchitecture or memory hierarchy simulators. Consequently, our approach is well suited to simulate samples spread over a large application since most of the native instructions are expected to execute in fast mode.

In the next section, we detail the approach of combining light static code annotation and instruction-set emulation. Section 3 presents our static code annotation tool: **calvin2**, and DICE, our instruction-set emulator. Section 4 evaluates the performance of **calvin2** + DICE, and compares it to Shade [2]. Section 5 summarizes this study and presents directions for future development.

2 Light Static Code Annotation and Instruction-Set Emulation

To trace programs or to perform on-the-fly simulations, static code annotation is generally more efficient than instruction-set emulation [7]. However, instruction-set emulation is a much more flexible approach: 1) implementing different tracing/simulation strategies is done without (re)instrumenting the target programs and 2) dynamically linked code, dynamically compiled code, and self-modifying code are traced and simulated easily.

Our approach takes advantage of the efficiency of static code annotation: a light instrumentation provides target programs with a fast (direct) execution mode which is used to rapidly position the execution in interesting execution sections. On the other hand, an instruction-set emulator is used to actually trace the target program or enable on-the-fly simulations. The emulator is embedded in the target program to be able to take control during the execution.

At run time, the program switches from the fast mode to the emulation mode whenever a *switching event* happens. The inserted annotation code only tests whether a switching event has occurred; on such an event, control is given to the emulator. Switching back from the emulation mode to the fast mode is managed by the emulator and is possible at any moment.

3 calvin2 and DICE

3.1 calvin2

calvin2 is a static code annotation tool derived from **calvin** [4], which instruments SPARC assembly code using SALTO [6]. **calvin2** lightly instruments the target programs by inserting *checkpoints*. The checkpoint code sequence consists in a few instructions (about 10) which checks whether the control has to be given to DICE, the emulator. We call the direct execution of the instrumented application the *fast execution mode*, as we expect this mode to exhibit a performance close to the original code performance. Switching from the fast mode to the emulation mode is triggered by a *switching event*.

Checkpoint layout. The execution overhead in fast mode is directly related with the number of inserted checkpoints. Checkpoints must not be too numerous. But, their number and distribution among the code executed determine the dynamic accuracy of mode switching (fast mode to emulation mode). In [3], we showed that inserting checkpoints at each procedure call and inside each path of a loop is a good tradeoff.

Switching Events. We call *switching event*, the event that, during the execution in fast mode, makes the execution switch to the emulation mode (switching back to the fast mode is managed by the emulator). Four different types of switching event have been implemented so far and are presented in [3]. Note that different switching events incur different overheads since the associated checkpoint code differs. In this paper, numerical results on the fast mode overhead are averaged upon the four switching event types implemented.

3.2 DICE: A Dynamic Inner Code Emulator

DICE emulates SPARC V9 instruction-set architecture (ISA) code using the traditional fetch-decode-interpret loop. DICE is an archive library linked with the target application. As such, it can receive the control, and return to direct execution at any moment during the execution by saving/restoring the host processor state. DICE works with programs instrumented by **calvin2**: the inserted checkpoints are used to give control to it.

DICE enables simulation by calling a user-defined analysis routine for each emulated instruction. Analysis routines have direct access to all information in the target program state, including complete memory state, and register values. DICE internals (emulation core, processor modeled, executable memory image, operating system interface, and user interface) are widely detailed in [3].

4 Performance Evaluation

In order to evaluate execution slowdowns incurred by both execution modes (fast mode and emulation mode), we gathered execution times on the SPEC95 benchmarks, running them entirely either in fast mode (with *ref* input data sets), or in emulation mode (with reduced *train* input data sets). In emulation mode, instruction and data references were traced. We compared **calvin2**+DICE to the Shade simulator [2]. Numerical results are presented in Table 1. The overhead measured in the "Shade WT" (Without Trace) column of Table 1 is the overhead needed to simulate the tested programs with the tracing capabilities enabled, but without actually tracing. This overhead can be viewed as the Shade "fast mode" overhead.

To simulate a complete microprocessor, an *additional* slowdown of, say, 1000 in emulation mode is still optimistic [1]. Given a one hour workload, and a low sampling ratio of, say, 1 %, using data from Table 1, we estimate that the simulation with Shade would require about $0.99 \times 17.07 + 0.01 \times (1000 + 82.19) =$

	Fast mode		Emulation mode	
	calvin2	**Shade WT**	**DICE**	**Shade**
CINT95 Avg.	1.60	21.63	119.54	87.04
CFP95 Avg.	1.21	12.90	115.57	77.76
SPEC95 Avg.	**1.31**	**17.07**	**117.47**	**82.19**

Table1. SPEC95 fast mode and emulation mode execution slowdowns.

27.72 hours; **calvin2** + DICE would take about $0.99 \times 1.31 + 0.01 \times (1000 + 117.45) = 12.47$ hours.

5 Summary and Future Work

In this paper, we have presented a new approach for running on-the-fly architecture simulations which combines light static code annotation and instruction-set emulation. The light static code annotation provides target programs with a *fast* (direct) *execution mode*. An *emulation mode* is managed by an embedded instruction-set emulator; it makes tracing/simulation possible. At runtime, the target program can dynamically switch between both modes.

We implemented a static code annotation tool, called **calvin2**, and an instruction-set emulator called DICE for the SPARC V9 ISA. We evaluated the performance of both tools, and compared it with the state of the art in dynamic translation, namely Shade. Running the SPEC95 benchmarks, the average fast mode execution slowdown has been 1.31. This makes it possible to skip large portions of long running workloads before entering the emulation mode to begin a simulation. Moreover, to simulate a small part of the execution, like this would be done in practice using long running workloads, **calvin2** + DICE are better suited than a tool like Shade.

Enabling complete execution-driven simulations with DICE is one of our main concerns. In addition, DICE has been extended to be embedded in a Linux kernel operating system. We are working on this extension of DICE, called LiKE (Linux Kernel Emulator), to make it simulate most of the operating system activity.

References

[1] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS'94*, May 1994.

[3] T. Lafage and A. Seznec. Combining light static code annotation and instruction-set emulation for flexible and efficient on-the-fly simulation. Technical Report 1285, IRISA, December 1999. `ftp://ftp.irisa.fr/techreports/1999/PI-1285.ps.gz`.

[4] T. Lafage, A. Seznec, E. Rohou, and F. Bodin. Code cloning tracing: A "pay per trace" approach. In *Euro-Par'99*, Toulouse, August 1999.

[5] S. Laha, J. Patel, and R. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[6] E. Rohou, F. Bodin, and A. Seznec. SALTO: System for assembly-language transformation and optimization. In *Proceedings of the Sixth Workshop Compilers for Parallel Computers*, December 1996.

[7] R. Uhlig and T. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 1997.

SCOPE - The Specific Cluster Operation and Performance Evaluation Benchmark Suite

Panagiotis Melas and Ed J. Zaluska

Electronics and Computer Science
University of Southampton, U.K.

Abstract. Recent developments in commodity hardware and software have enabled workstation clusters to provide a cost-effective HPC environment which has become increasingly attractive to many users. However, in practice workstation clusters often fail to exploit their potential advantages. This paper proposes a tailored benchmark suite for clusters called Specific Cluster Operation and Performance Evaluation (SCOPE) and shows how this may be used in a methodology for a comprehensive examination of workstation cluster performance.

1 Introduction

The requirements for High Performance Computing (HPC) have increased dramatically over the years. Recent examples of inexpensive workstation clusters, such as the Beowulf project, have demonstrated cost-effective delivery of high-performance computing (HPC) for many scientific and commercial applications. This establishment of clusters is primarily based on the same hardware and software components used by the current commodity computer "industry" together with parallel techniques and experience derived from from Massively Parallel Processor (MPP) systems. As these trends are expected to continue in the foreseeable future, workstation cluster performance and availability is expected to increase accordingly.

In order for clusters to be established as a parallel platform with MPP-like performance, issues such as internode communication, programming models, resource management and performance evaluation all need to be addressed [4]. Prediction and performance evaluation of clusters is necessary to assess the usefulness of current systems and provide valuable information to design better systems in the future.

This paper proposes a performance evaluation benchmark suite known as the Specific Cluster Operation and Performance Evaluation (SCOPE) benchmark suite. This benchmark suite is designed to evaluate the potential characteristics of workstation clusters as well as providing developers with a comprehensive understanding of the performance behaviour of clusters.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 183–188, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Performance Evaluation of HPC Systems and Clusters

Clusters have emerged as a parallel platform with many similarities with MPPs but at the same time strong quantitative and qualitative differences from other parallel platforms. MPPs still have several potential advantages over clusters of workstations. The size and the quality of available resources per node is in favour of MPPs, e.g. communication and I/O, subsystems, memory hierarchy. In MPP systems software is highly optimised to exploit the underlying hardware fully while clusters use general-purpose software with little optimisation.

Despite the use of Commodity Off The Shelf (COTS) components the classification of clusters of workstations is somewhat loose and virtually every single cluster is built with its own individual architecture and configuration reflecting the nature of each specific application. Consequently there is a need to examine closer the performance behaviour of the interconnect network for each cluster. Existing HPC benchmark suites for message-passing systems, such as PARKBENCH and NAS benchmarks, already run on clusters of workstations but only because clusters support the identical programming model as the MPP systems these benchmarks were written for [1, 2]. Although the above condition is theoretically sufficient for an MPP benchmark to run on a workstation cluster ("how much"), it does not necessary follow that any useful information or understanding about specific performance characteristics of clusters of workstations will be provided. This means that the conceptual issues underlying the performance measurement of workstation clusters are frequently confused and misunderstood.

3 The Structure of the SCOPE Benchmark

The concept of this tailored benchmark suite is to measure the key information to define cluster performance, Following EuroBench and PARKBENCH [3, 5] methodology and re-using existing benchmark software where possible. SCOPE tests are classified into single-node-level performance, interconnection-level performance and computational model level performance (Table 1).

Single node tests are intended to measure the performance of a single node-workstation of a cluster, (these are also known as *basic architectural benchmarks*) [3, 5]. Several well-established benchmarks such as LINPACK, or SPEC95 are used here as they provide good measures of single-node hardware performance.

In order to emphasise the importance of internode communication in clusters, the SCOPE low-level communication tests include additional network-level tests to measure the "raw" performance of the interconnection network. Optimisation for speed is a primary objective of low-level tests using techniques such as cache warm-up and page alignment. Performance comparisons through these levels provide valuable information within the multilayered structure of typical cluster subsystems. Latency and bandwidth performance can be expressed as a function of the message size and Hockney's parameters r_∞ and $n_{1/2}$ are directly applicable.

Collective communication routines are usually implemented on top of single peer-to-peer calls, therefore their performance is based on the efficiency of the

Table 1. The structure of the SCOPE suite

Test Level		Test Name	Comments
SINGLE NODE		LINPACK, SPEC95, etc	Existing tests
LOW-LEVEL	Network level	Latency/Bandwidth	Pingpong-like
	Message-passing	Latency/Bandwidth	Pingpong-like
	Collective	Synchronise	Barrier test
		Broadcast	Data movement
		Reduce	Global comput.
		All-to-all	Data movement
KERNEL-LEVEL	Operation	Shift operation	Send-Recv-like
		Gather operation	Vectorised op.
		Scatter operation	Vectorised op.
		Broadcast operation	Vectorised op.
	Algorithmic	Matrix multiplication	Row/Column
		Relaxation algorithm	Gauss-Seidel
		Sorting algorithm	Sort (PSRS)

algorithm implemented (e.g. binomial tree), peer-to-peer call performance, group size (p) and the underlying network architecture. Collective tests can be divided into three sub-classes: synchronisation (i.e. barrier call), data movement (i.e. broadcast and all-to-all) and global computation (i.e. reduce operation call).

Traditionally kernel-level tests use algorithms or simplified fractions of real applications. The SCOPE kernel-level benchmarks also utilise algorithmic and operation tests. Kernel-level operation tests provide information on the delivered performance at the kernel-level of fundamental message passing operations such as broadcast, scatter and gather.

This section of the benchmark suite makes use of a small set of kernel-level algorithmic tests which are included in a wide range of real parallel application algorithms. Kernel-level algorithmic tests will measure the overall performance of a cluster at a higher programming level. The kernel-level algorithms included at present in SCOPE are matrix-matrix multiplication algorithms, a sort algorithm (Parallel Sort with Regular Sampling) and a 2D-relaxation algorithm (mixed Gauss-Jacobi/Gauss-Seidel). A particular attribute of these tests is the degree in which they can be analysed and their provision of elementary level performance details which can be used to analyse more complicated algorithms later.

In addition, other kernel-level benchmark tests such as the NAS benchmarks can also be used as part of the SCOPE kernel-level tests to provide application-specific performance evaluation. In a similar way the SCOPE benchmarks (excluding the low-level network tests) can also be used to test MPP systems.

4 Case Study Analysis and Results

This section demonstrates and briefly analyses the SCOPE benchmark results obtained with our experimental SCOPE implementation on a 32-node beowulf cluster at Daresbury and the 8-node Problem Solving Environment beowulf cluster at Southampton. The architecture of the 32-node cluster is based on Pentium III 450 MHz CPU boards and is fully developed and optimised. On the other hand, the 8-node cluster is based on AMD Athlon 600MHz CPU boards and is still under development. Both clusters use a dedicated 100 Mbit/sec network interface for node interconnection, using the MPI/LAM 6.3.1 communication library under Linux 2.2.12.

Table 2. SCOPE benchmark suite results for Beowulf clusters

SCOPE Test		Daresbury 32-node cluster			PSE 8-node cluster		
Network latency, BW		63 μs	10.87 MB/s		47.5 μs	10.95 MB/s	
MPI latency, BW		74 μs	10.86 MB/s		60.3 μs	10.88 MB/s	
Collective tests							
	Size	2-node	4-node	8-node	2-node	4-node	8-node
Synch	-	150 μs	221 μs	430 μs	118 μs	176 μs	357 μs
Broadcast	1 KB	301 μs	435 μs	535 μs	243 μs	324 μs	495 μs
Reduce	1 KB	284 μs	302 μs	866 μs	245 μs	258 μs	746 μs
All-to-all	1 KB	293 μs	580 μs	1102 μs	260 μs	357 μs	810 μs
Kernel-level tests (600X600 matrix)							
	Size	2-node	4-node	8-node	2-node	4-node	8-node
Bcast op.	600x600	0.277 s	0.552 s	1.650 s	0.337 s	0.908 s	1.856 s
Scatter op.	600x600	0.150 s	0.229 s	0.257 s	0.212 s	0.239 s	0.245 s
Gather op.	600x600	0.163 s	0.265 s	0.303 s	0.192 s	0.243 s	0.266 s
Shift op.	600x600	0.548 s	0.572 s	0.573 s	0.617 s	0.615 s	0.619 s
	Size	2-node	4-node	8-node	2-node	4-node	8-node
Matrix	1080x1080	50.5 s	25.5 s	14.0 s	123 s	57.6 s	27.4 s
Relaxation	1022x1022	126 s	66.0 s	36.0 s	148 s	76.3 s	41.9 s
Sorting	8388480	78,8 s	47.9 s	24.9 s	46.4 s	29.6 s	21.8 s

The first section of Table 2 gives results for the SCOPE low-level tests, clearly the cluster with the fastest nodes give better results for peer-to-peer and collective tests. TCP/IP level latency is 47.5 μs for the PSE cluster and 63 μs for the Daresbury cluster while the effective bandwidth is around 10.9 MB/s for both clusters. The middle section of Table 2 presents the results for kernel-level operation tests for array sizes of 600x600 on 2, 4 and 8 nodes. The main difference between the low-level collective tests and the kernel-level operation tests

is the workload and the level at which performance is measured, e.g. the low-level tests exploit the use of cache, while kernel-level operations measure buffer initialisation as well. The picture now is reversed, the Daresbury cluster giving better results over the PSE cluster. Both clusters show good scalability for the scatter/gather and shift operation tests.

The last part of Table 2 presents results for the kernel-level algorithmic tests. The matrix multiplication test is based on the Matrix Row/Column Striped algorithm for 1080x1080 matrix size. The PSRS algorithm test sorts floating point vectors of size 8 million cell array. The multi-grid relaxation test presented is a mixture of Gauss-Jacobi and Gauss-Seidel iteration methods on a 1022x1022 array over 1000 iterations.

Results from these tests indicate a good (almost linear) scalability for the first two tests with a communication overhead. The implementation of the sort algorithm requires a complicated communication structure with many initialisation phases and has poor scalability. The performance difference between these clusters measured by the kernel-level tests demonstrates clearly the limited development of the PSE cluster which at the time of the measurements was under construction.

5 Conclusions

Workstation clusters using COTS have the potential to provide, at low cost, an alternative parallel platform suitable for many HPC applications.

A tailored benchmark suite for clusters called Specific Cluster Optimisation and Performance Evaluation (SCOPE) has been produced. The SCOPE benchmark suite provides a benchmarking methodology for the comprehensive examination of workstation cluster performance characteristics. An initial implementation of the SCOPE benchmark suite was used to measure performance on two clusters and the results of these tests have demonstrated the potential to identify and classify cluster performance.

Acknowledgements

We thank Daresbury Laboratory for providing access to their 32-node cluster

References

[1] F. Cappello, O. Richard, and D. Etiemble. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. *Lecture Notes in Computer Science*, 1662:339–348, 1999.

[2] John L. Gustafson and Rajat Todi. Conventional benchmarks as a sample of the performance spectrum. *The Journal of Supercomputing*, 13(3):321–342, May 1999.

[3] R. Hockney. *The Science of Computer Benchmarking.* SIAM, 1996.

[4] Dhabaleswar K. Panda and Lionel M. Ni. Special Issue on Workstation Clusters and Network-Based Computing: Guest Editors' introduction. *Journal of Parallel and Distributed Computing*, 40(1):1–3, January 1997.

[5] Adrianus Jan van der Steen. *Benchmarking of High Performance Computers for Scientific and Technical Computation*. PhD thesis, ACCU, Utrecht, Netherlands, March 1997.

Implementation Lessons of Performance Prediction Tool for Parallel Conservative Simulation

Chu-Cheow Lim[1], Yoke-Hean Low[2], Boon-Ping Gan[3], and Wentong Cai[4]

[1] Intel Corporation
SC12-305, 2000 Mission College Blvd, Santa Clara, CA 95052-8119, USA
chu-cheow.lim@intel.com
[2] Programming Research Group, Oxford University Computing Laboratory
University of Oxford, Oxford OX1 3QD, UK
mlow@comlab.ox.ac.uk
[3] Gintic Institute of Manufacturing Technology
71 Nanyang Drive, Singapore 638075, Singapore
bpooi@gintic.gov.sg
[4] Center for Advanced Information Systems, School of Applied Science
Nanyang Technological University, Singapore 639798, Singapore
aswtcai@ntu.edu.sg

Abstract. Performance prediction is useful in helping parallel programmers answer questions such as speedup scalability. Performance prediction for parallel simulation requires first working out the performance analyzer algorithms for specific simulation protocols. In order for the prediction results to be close to the results from actual parallel executions, there are further considerations when implementing the analyzer. These include (a) equivalence of code between the sequential program and parallel program, and (b) system effects (e.g. cache miss rates and synchronization overheads in an actual parallel execution). This paper describes our investigations into these issues on a performance analyzer for a conservative, "super-step" (*synchronous*) simulation protocol.

1 Introduction

Parallel programmers often use performance predictions to better understand the behavior of their applications. In this paper, we discuss the main implementation issues to consider in order to obtain accurate and reliable prediction results. The performance prediction study[1] is carried out on a parallel discrete-event simulation program for wafer fabrication models. Specifically, we designed a performance analyzer algorithm (also referred to as a parallelism analyzer algorithm), and implemented it as a module that can be "plugged" into a sequential simulation to predict performance of a parallel run.

[1] The work was carried out when the first two authors were with Gintic. The project is an on-going research effort between Gintic Institute of Manufacturing Technology, Singapore, and School of Applied Science, Nanyang Technological University, Singapore, to explore the use of parallel simulation in manufacturing industry.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 189–193, 2000.
© Springer-Verlag Berlin Heidelberg 2000

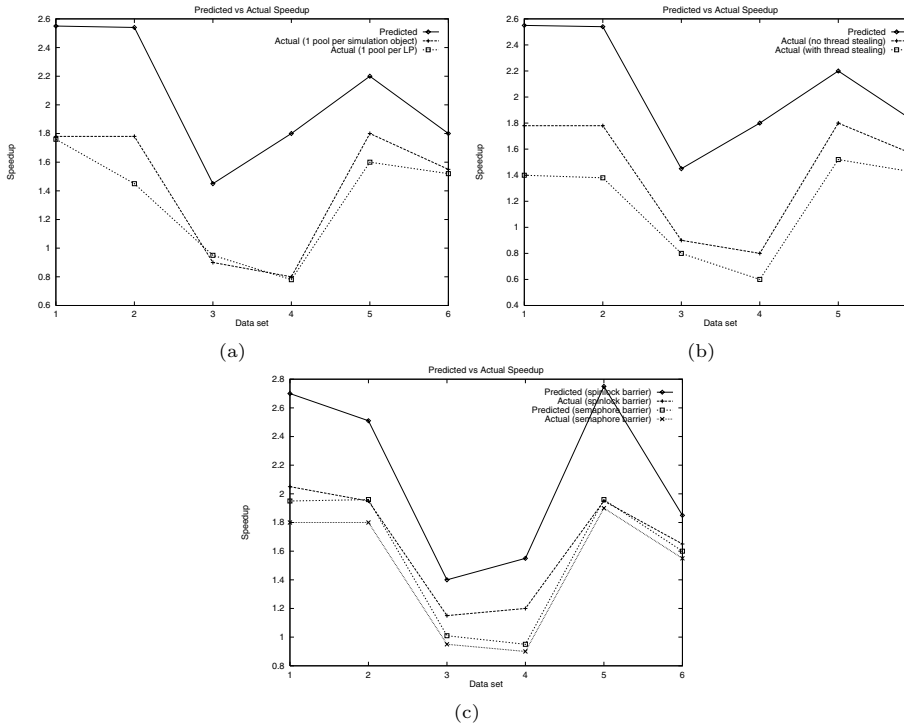


Fig. 1. Predicted and actual speedups for 4 processors using conservative synchronous protocol. (a) one event pool per simulation object and per LP. (b) thread-stealing turned on and turned off in the Active Threads library. (c) spinlock barrier and semaphore barrier.

2 Analyzer for Conservative Simulation Protocol

The parallel conservative super-step synchronization protocol and the performance analyzer for the protocol are described in [1]. The simulation model is partitioned into entities which we refer to as *logical processes* (LPs), such that LPs can only affect one another's state via event messages. In our protocol, the LPs execute in *super-steps*, alternating between execution (during which events are simulated) and barrier synchronization (at which point information is exchanged among the LPs).

The performance analyzer is implemented as a "plug-in" module to a sequential simulator. The events simulated in the sequential simulator are fed into the performance analyzer to predict the performance of the parallel conservative super-step protocol as the sequential simulation progresses.

Our simulation model is for wafer fabrication plants and is based on the Sematech datasets [3]. The parallel simulator uses the Active Threads library [4]

as part of its runtime system. A thread is created to run each LP. Multiple simulation objects (e.g. machine sets) in the model are grouped into one LP. Both the sequential and parallel simulators are written in C++ using GNU g++ version 2.7.2.1. Our timings were measured on a four-processor Sun Enterprise 3000, 250 MHz Ultrasparc 2.

3 Issues for Accurate Predictions

Code equivalence and system effects are two implementation issues to be considered in order to obtain accurate results from the performance analyzer. To achieve code equivalence between the parallel simulator and the sequential simulator, we had to make the following changes:

(1) To get the same set of events as in a parallel implementation, the sequential random number generator in the sequential implementation is replaced by the parallel random number generator used in the parallel implementation.
(2) A technique known as *pre-sending* [2] is used in the parallel simulation to allow for more parallelism in the model. If the sequence of events in the sequential simulator is such that event E_1 generates $E2$ which in turn generates $E3$, pre-sending may allow event E_1 to generate E_2 and E_3 simultaneously. The event execution time of E_1 with pre-sending may be different from the non-presend version. We modified the sequential simulator to use the pre-sending technique.
(3) Our sequential code has a global event pool manager managing event objects allocation and deallocation. The parallel code initially has one local event pool manager associated with each simulation object. We modified the parallel code to have one event pool for each LP. (A global event pool would have introduced additional synchronization in the parallel code.) Table 1a shows that external cache miss rates for datasets 2 and 4 are reduced. The corresponding speedups are also improved (Figure 1a). The actual speedup with one event pool per LP is now closer to the predicted trend.

There are two sources of system effects that affect the performance of a parallel execution:(a) cache effects (b) synchronization mechanisms used.

Cache effects Our parallel implementation uses the Active Threads library [4] facilities for multi-threading and synchronization. The library has a self load-balancing mechanism in which an idle processor will look into another processor's thread queue and, if possible, try to "steal" a thread (and its LP) to run. Disabling the thread-stealing improves the cache miss rates (Table 1b) and brings the actual speedup curve closer to the predicted one (Figure 1b).

Program synchronization At the end of each super-step in the simulation protocol, all the LPs are required to synchronize at a barrier. We experimented with two barrier implementations: (a) using semaphore and (b) using spinlock.

Data set	1	2	3	4	5	6
Parallel (One event pool per simulation object)	6.4%	8.4%	10.9%	12.2%	7.2%	7.1%
Parallel (One event pool per LP)	6.0%	6.3%	11.9%	12.2%	5.9%	7.3%
Sequential	0.36%	0.29%	0.18%	0.12%	0.50%	0.61%

(a)

Data set	1	2	3	4	5	6
Parallel (1 event pool per LP, thread stealing on)						
References	373.8×10^6	820.7×10^6	1627.1×10^6	83.9×10^6	368.4×10^6	344.4×10^6
Hits	332.5×10^6	735.4×10^6	1378.1×10^6	69.5×10^6	329.2×10^6	305.9×10^6
Miss %	11.1	10.4	15.3	17.2	10.6	11.2
Parallel (1 event pool per LP, thread stealing off)						
References	379.3×10^6	730.2×10^6	1736.3×10^6	79.3×10^6	328.5×10^6	382.0×10^6
Hits	356.6×10^6	684.1×10^6	1529.3×10^6	69.7×10^6	309.0×10^6	354.0×10^6
Miss %	6.0	6.3	11.9	12.2	5.9	7.3

(b)

Table 1. External cache miss rates for (a) parallel implementation using one event pool per simulation object, one event pool per LP, and sequential execution; (b) parallel implementation when thread-stealing mechanism in the runtime system is turned on or off.

We estimated the time of each barrier from separate template programs. The estimate is 35 μs for a "semaphore barrier" and 6 μs for a "spinlock barrier". The total synchronization cost is obtained from multiplying the per-barrier time by the number of super-steps that each dataset uses. Figure 1c shows the predicted and actual speedup curves (with thread-stealing disabled) for both barriers. The predicted and actual speedup curves for "semaphore barrier" match quite closely. There is however still a discrepancy for the corresponding curves for "spinlock barrier". Our guess is that the template program under-estimates the time for a "spinlock barrier" in a real parallel program.

4 Conclusion

This paper describes the implementation lessons learnt when we tried to match the trends in a predicted and an actual speedup curve on a 4-processor Sun shared-memory multiprocessor. The main implementation issue to take note of is that the code actions in a sequential program should be comparable to what would actually happen in the corresponding parallel program. The lessons learnt was put to good use in implementing a new performance analyzer for a conservative, *asynchronous* protocol. Also, in trying to get the predicted and actual curves to match, the parallel simulation program was further optimized.

References

[1] C.-C. Lim, Y.-H. Low, and W. Cai. A parallelism analyzer algorithm for a conservative super-step simulation protocol. In *Hawaii International Conference on System Sciences (HICSS-32)*, Maui, Hawaii, USA, January 5–8 1999.

[2] D. Nicol. *Problem characteristics and parallel discrete event simulation*, volume Parallel Computing: Paradigms and Applications, chapter 19, pages 499–513. Int. Thomson Computer Press, 1996.

[3] Sematech. Modeling data standards, version 1.0. Technical report, Sematech, Inc., Austin, TX78741, 1997.

[4] B. Weissman and B. Gomes. Active threads: Enabling fine-grained parallelism. In *Proceedings of 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, Nevada USA, July 13 – 16 1998.

A Fast and Accurate Approach to Analyze Cache Memory Behavior

Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo

Computer Architecture Department
Universitat Politècnica de Catalunya-Barcelona
{xvera, josepll, antonio, nbermudo}@ac.upc.es

Abstract. In this work, we propose a fast and accurate approach to estimate the solution of Cache Miss Equations (CMEs), which is based on the use of sampling techniques. The results show that only a few seconds are required to analyze most of the SPECfp benchmarks with an error smaller than 0.01.

1 Introduction

To take the best advantage of caches, it is necessary to exploit as much as possible the locality that exists in the source code. A locality analysis tool is required in order to identify the sections of the code that are responsible for most penalties and to estimate the benefits of code transformations. Several methods, such as simulators or compilers heuristics, can estimate this information. Unfortunately, simulators are very slow whereas heuristics can be very imprecise.

Our proposal is based on estimating the result of Cache Miss Equations (CMEs) by means of sampling techniques. This technique is very fast and accurate, and the confidence of the error can be chosen.

2 Overview of CMEs

CMEs [4] are an analysis framework that describes the behavior of a cache memory. The general idea is to obtain for each memory reference a set of constraints and equations defined over the iteration space that represent the cache misses. These equations make use of the reuse vectors [9]. Each equation describes the iteration points where the reuse is not realized. For more details, the interested reader is referred to the original publications [4, 5].

2.1 Solving CMEs

There are two approaches in order to solve CMEs, solve them analitically [4] or checking whether an iteration point is a solution or not. The first approach only works for direct-mapped caches, whereas the second can be used for both direct-mapped and set-associative organizations.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 194–198, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Traversing the iteration space Given a reference, all the iteration points are tested independently, studying the equations in order: from the equations generated by the shortest reuse vector to the equations generated by the longest one [5]. For this approach, we need to know whether a polyhedron is empty after substituting the iteration point in the equation. This is still a NP-Hard problem, however only $s * number\_of\_points$ polyhedra must be analyzed.

3 Sampling

Our proposal builds upon the second method to solve the CMEs (traversing the iteration space). This approach to solve the CMEs allows us to study each reference in a particular iteration point independently of all other memory references. Based on this property, a small subset of the iteration space can be analyzed, reducing heavily the computation cost. In particular, we use random sampling to select the iteration points to study, and we infer the global miss ratio from them. This sampling technique cannot be applied to a cache simulator. A simulator cannot analyze an isolated reference, since it requires information of all previous references.

3.1 CMEs Particularization

We represent a perfectly nested loop of depth n with known bounds as a finite convex polyhedron of the n-dimensional iteration space $\mathbb{Z}^n$. We are interested in finding the number of misses this loop nest produces (said m). In order to obtain it, for each reference belonging to the loop nest we define a random variable (RV) that returns the number of misses. Below, we show that this RV follows a Binomial distribution. Thus, we can use statistical techniques (see the full paper[1] [8]) to compute the parameters that describe it.

For each memory instruction, we can define a Bernoulli-RV $X \sim B(p)$ as follows:

$$\begin{aligned} X : Iteration \quad & Space \longrightarrow \mathbb{R} \\ \boldsymbol{\imath} \quad & \longrightarrow \{0, 1\} \end{aligned}$$

such that $X \equiv 1$ if the memory instruction results in a miss for iteration $\boldsymbol{\imath}$, $X \equiv 0$ otherwise. Note that X describes the experiment of choosing an iteration point and checking whether the memory instruction produces a miss for it, and p is the probability of success. The value of p is $p = \frac{m}{N}$, where N is the number of iteration points.

Then, we repeat N times the experiment, using a different iteration point in each experiment, obtaining $X_1, \ldots, X_N$ different RV-variables. We note that:

- All the $X_i, i = 1 \ldots N$ have the same value of p.
- All the $X_i, i = 1 \ldots N$ are independent.

[1] ftp://ftp.ac.upc.es/pub/reports/DAC/2000/UPC-DAC-2000-8.ps.Z

The variable $Y = \sum X_i$ represents the total number of misses in all N experiments. This new variable follows a binomial distribution with parameters Bin(N,p) [3] and it is defined over all the iteration space. By generating random samples over the iteration space, we can infer the total number of misses.

3.2 Generating Samples

The key issues to obtain a good sample are:

- It is important that all the population is represented in the sample.
- The size of the sample.

In our case, we have to keep in mind another constraint: the sample cannot have repeated iteration points (one iteration point cannot result in a miss twice).

To fulfill these requirements, we use *Simple Random Sampling* [6]. The size of the sample is set according to the required width of the confidence interval and the desired confidence. For instance, for an interval width of 0.05 and a 95% confidence, 1082 iteration points has to be analyzed.

4 Evaluation

CMEs have been implemented for fortran codes through the Polaris Compiler [7] and the Ictineo library [1]. We have used our own polyhedra representation [2]. Both direct-mapped and set-associative caches with LRU replacement policy are supported.

4.1 Performance Evaluation

Next, we evaluate the accuracy of the proposed method and the speed/accuracy trade-offs for both direct-mapped and set-associative caches.

The loop nests considered are obtained from the SPECfp95, choosing for each program the most time consuming loop nests that in total represent between the 60-70% of the execution time using the reference input data. The simulation values are obtained using a trace driven cache simulator by instrumenting the program with Ictineo. For the evaluation of the execution time, an Origin2000 has been used.

The CMEs have been generated assuming a 32KB cache of arbitrary associativity.

From our experiments we consider that a 95% confidence and an interval of 0.05 is a good trade-off between analysis time and accuracy, since the programs are usually analyzed in a few seconds, and never more than 2 minutes. The more accurate configurations require more time because more points are considered.

With a 95% confidence and an interval width of 0.05, the absolute difference between the miss ratio and the central point of the confidence interval is usually lower than 0.002 and never higher than 0.01.

Set-Associative Caches

We have also analyzed the SPECfp programs for three different organizations of set-associative caches (2-way, 4-way and 8-way). Although in general the analysis time is higher than for direct mapped caches, it is reasonable (never more than 2 minutes).

As the case of direct mapped caches, the difference between the miss ratio and the empirical estimation for all the programs is usually lower than 0.002 and never higher than 0.01.

5 Conclusions

In this paper we propose the use of sampling techniques to solve CMEs. With these techniques we can perform memory analysis extremely fast independently of the size of the iteration space. For instance, it takes the same time (3 seconds) to analyze a matrix by matrix loop nest of size 100x100 than one of size 1000x1000. On the contrary it takes 9 seconds to simulate the first case and more than 2 hours to simulate the second.

In our experiments we have found that, using a 95% confidence and an interval width of 0.05, the absolute error in the miss ratio was smaller than 0.002 in 65% of the loops from the SPECfp programs and was never bigger than 0.01. Furthermore, the analysis time for each program was usually just a few seconds and never more than 2 minutes.

Acknowledgments

This work has been supported by the ESPRIT project MHAOTEU (EP 24942) and the CICYT project 511/98.

References

[1] Eduard Ayguadé et al. *A uniform internal representation for high-level and instruction-level transformations.* UPC, 1995.
[2] Nerina Bermudo, Xavier Vera, Antonio González, and Josep Llosa. An efficient solver for cache miss equations. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.
[3] M.H. DeGroot. *Probability and statistics.* Addison-Wesley, 1998.
[4] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS97*, 1997.
[5] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *ASPLOS98*, 1998.
[6] Moore; McCabe. *Introduction to the Practice of Statistics.* Freeman & Co, 1989.
[7] David Padua et al. *Polaris developer's document*, 1994.

[8] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior. Technical Report UPC-DAC-2000-8, Universitat Politècnica de Catalunya, February 2000.
[9] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN91*, 1991.

Impact of PE Mapping on Cray T3E Message-Passing Performance

Eduardo Huedo, Manuel Prieto, Ignacio M. Llorente, and Francisco Tirado

Departamento de Arquitectura de Computadores y Automática
Facultad de Ciencias Físicas
Universidad Complutense
28040 Madrid, Spain
Phone: +34-91 394 46 25
Fax +34-91 394 46 87
{ehuedo, mpmatias, llorente, ptirado}@dacya.ucm.es

Abstract. The aim of this paper is to study the influence of processor mapping on message passing performance of two different parallel computers: the Cray T3E and the SGI Origin 2000. For this purpose, we have first designed an experiment where processors are paired off in a random manner and messages are exchanged between them. In view of the results of this experiment, it is obvious that the physical placement must be accounted for. Consequently, a mapping algorithm for the Cray T3E, suited cartesian topologies is studied. We conclude by making comparisons between our T3E algorithm, the MPI default mapping and another algorithm proposed by Müller and Resch in [9].

Keywords. MPI performance evaluation, network contention, mapping algorithm, Cray T3E, SGI Origin 2000.

1. Introduction

The belief has spread that processor mapping does not have any significant effect on modern multiprocessor performance. Consequently, this is no cause of concern to the programmer, who only has to distinguish between local and remote access. It is true that since most parallel systems today use wormhole or cut-through routing mechanisms, the message *first bit delay* is nearly independent of the number of hops that it must travel through. However, message latency depends also on the network state when the message is injected into it. Indeed, as we have shown in [1], *blocking times*, i.e. delays due to conflicts over the use of the hardware routers and communication links, are the major contributors to message latencies under heavy or unevenly distributed network traffic.

Consequently, as shown in section 2, a correct correspondence between logical and physical processor topologies could improve performance, since it could help to reduce network contention. In section 3 we propose a mapping algorithm that can be used to optimize the `MPI_Cart_create` function in the Cray T3E. Finally, some experimental results and conclusions are presented in sections 4 and 5 respectively.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 199-207, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2. Random Pairwise Exchanges

In order to illustrate the importance of processor mapping, we have performed an experiment in which all the available processors in the system pair off in a random manner and exchange messages using the following scheme:

```
MPI_Isend(buf1, size, MPI_DOUBLE, pair, tag1, Comm, &sreq);
MPI_Irecv(buf2, size, MPI_DOUBLE, pair, tag2, Comm, &rreq);
MPI_Wait(&sreq, &sstatus);
MPI_Wait(&rreq, &rstatus);
```

As we will show, for the parallel systems studied, an important degradation in performance is obtained when processors are poorly mapped.

2.1. Random Pairing in the Cray T3E

The Cray T3E in which the experiments of this paper have been carried out is made up of 40 Alpha 21264 processors running at 450MHz [3][4].

Figure 1 shows the results obtained using 32 processors. This system has two communication modes: messages can either be *buffered* by the system or be sent directly in a *synchronous* way. The best unidirectional bandwidth attainable with synchronous mode (using the typical ping-pong test) is around 300 MB/s, while using the buffered mode it is only half the maximum (around 160MB/s) [1][4][5].

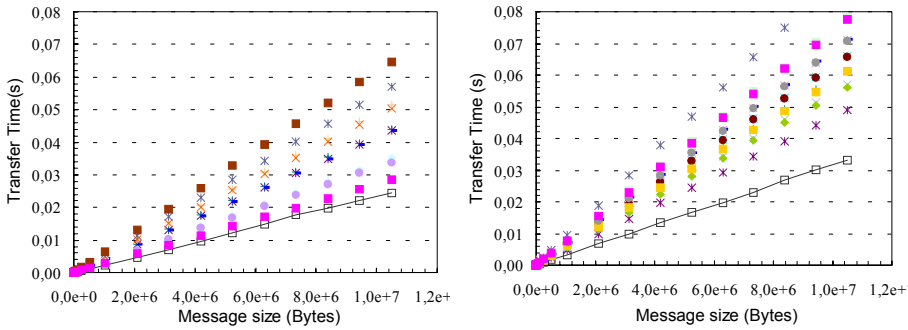


Fig. 1. Transfers times (seconds) in the T3E obtained from different pairings. The measurement has been taken using buffered (right-hand chart) and synchronous (left-hand chart) mode.

For medium messages the difference between the best and worst pairing is significant. For example, the optimal mapping is approximately 2.7 times better than the worst for a 10MB message size in both communications modes.

It is also interesting to note that for the optimal mapping, the bandwidth of this experiment is better than the unidirectional one, due to the exploitation of the T3E bi-directional links. In the synchronous mode the improvement is approximately 40% (around 420MB/s) reaching almost 100% in the buffered one (around 300MB/s).

Figure 3 shows the usual correspondence between the physical and logical processors for a 4x4 topology and one of the possible optimal mapping, where every neighbour pair in the logical topology share a communication link in the physical one:

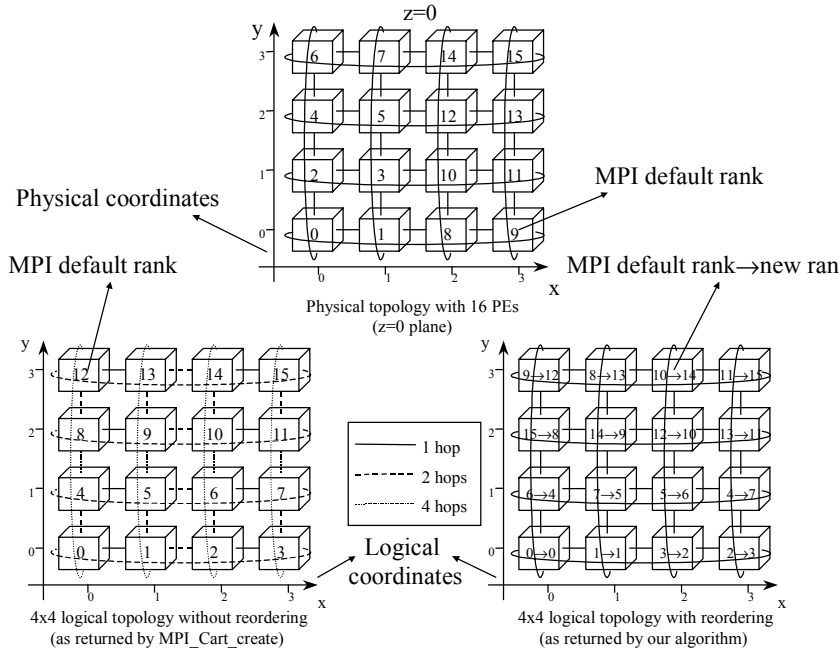


Fig. 2. Correspondence between physical and logical mapping in the Cray T3E.

2.2. Random Pairing in the SGI Origin 2000

Figure 2 shows the results obtained on the SGI Origin 2000. The system studied in this paper consists of 32 R10000 microprocessors running at 250 MHz [4][6].

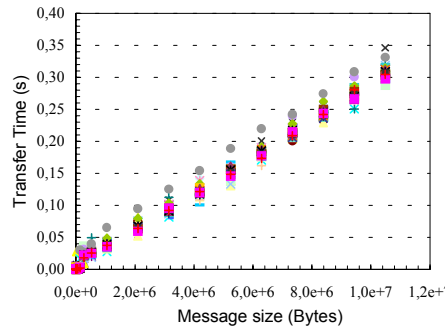


Fig. 3. Transfer times (seconds) in the SGI Origin 2000 for different message sizes obtained from different pairings.

Here, the behavior is different: performance depends on the pairing but the difference between the best and the worst is only around 20% and does not grow with message size. In addition, when comparing to the unidirectional bandwidth there is no improvement. On the contrary, performance is worse for message sizes larger than 126KB. The effective bandwidth is only around 40 MB/s, while using a ping-pong test a maximum of around 120 MB/s can be obtained (for 2 MB messages)[1][2][7].

2.3. Preliminary Conclusions

In view of these results it is obvious that the physical placement must be accounted for. On the SGI Origin 2000 it is possible to perform the mapping control by combining MPI with an additional tool for data placement, called `dplace` [8]. Unfortunately, to the best of the author's knowledge, nothing similar exist on the T3E.

For n-dimensional cartesian topologies, the MPI standard provides for this need through the function `MPI_Cart_create`. It contains one option that allows the processor rank to be reordered with the aim of matching the logical and physical topologies to the highest possible degree. However, the MPI on the Cray does not carry out this optimization.

3. `MPI_Cart_create` Optimization on the Cray T3E

The only attempt, to our knowledge, to improve the implementation of the `MPI_Cart_create` function on the Cray T3E is a heuristic processor-reordering algorithm proposed by M. Müller and M. Resch in [9]. This algorithm sorts PE ranks according to physical coordinates using the 6 permutations of the (x, y, z) triplet and then selects the optimum with respect to the average hop count. Although this technique improves the MPI default mapping, we have in most cases obtained better results using an alternative approach based on a greedy algorithm.

3.1. Our Mapping Algorithm

The aim is to find a rank permutation that minimizes the distance between (logical) neighbouring processing elements (PEs). We shall start by making some definitions:

- *Physical distance* (d_p): the minimum number of hops one message must travel from the origin PE to the destination PE.
- *Logical distance* (d_l): the absolute value of the PE rank difference.

According to these definitions, the *distance* (d) used by our algorithm is:

$$d(PE, PE_1) < d(PE, PE_2) \Leftrightarrow \quad \textbf{(1)}$$
$$\begin{cases} d_p(PE, PE_1) < d_p(PE, PE_2) \\ \vee \\ (d_p(PE, PE_1) = d_p(PE, PE_2)) \wedge (d_l(PE, PE_1) < d_l(PE, PE_2)) \end{cases}$$

The mapping evaluation function, which has to be minimized by the algorithm, consists in calculating the *average distance*:

$$d_{av} = \frac{\sum_{i \in \{PEs\}} \sum_{j \in neighbours(i)} d_p(i,j)}{\sum_{i \in \{PEs\}} \#neighbours(i)} \quad \textbf{(2)}$$

In the optimal case d_{av} is equal to 1. However, it may or may not be obtained depending on the cartesian topology chosen and the system configuration and state.

To find out the *physical distance* between PEs we use the physical co-ordinates of each PE (obtained with `sysconf(_SC_CRAY_PPE)`, which does not form part of MPI) and the knowledge of the Cray T3E bi-directional 3D-torus physical topology.

3.2. 1D Algorithm

The following scheme describes our 1D-mapping algorithm. Starting from a given processor, each step consists of two phases. First, the PE at minimum distance from the current one is chosen and then appended to the topology:

```
1D_algorithm(int dim, int current) {
  append(current);
  for (i= 1; i<dim; i++) {
    next= (not used) PE at minimum d from current;
    append(next);
    current= next;
  }
}
```

The algorithm is written in such way that it is possible to choose the first PE. For example, if the number of PEs is even it should start from the rank 0 PE. Otherwise, it should start from the highest rank PE. Figure 4 illustrates the algorithm in both cases:

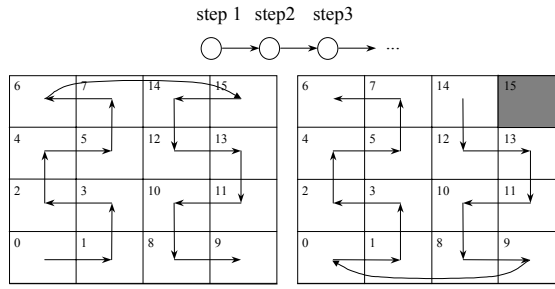


Fig. 4. Mapping of our 1D algorithm for 16 (left-hand chart) and 15 (right-hand chart) PEs.

3.3. N-Dimensional Algorithm

The generalization to an n-dimensional algorithm is built up from the previous one:

```
algorithm(int ndim, int dim[], int current) {
  if (ndim==1) /* 1D algorithm starting from current */
    1D_algorithm(dim[0],current);
  else {        /* Recursive algorithm */
    algorithm(ndim-1,dim,current);
    for (i= 1; i<dim[ndim-1]; i++) {
      next= (not used) PE at minimum d from current;
      algorithm(ndim-1,dim,next);
      current= next;
    }
  }
}
```

In the case that one dimension was even and the other was odd, it would be better to use first the even one. The choice of the first PE is done as in the 1D-algorithm. Figure 5 illustrates the algorithm in two cases:

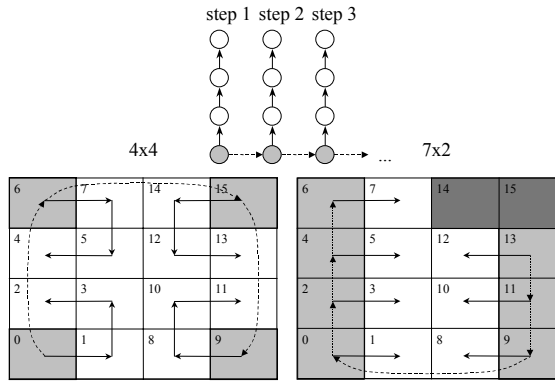


Fig. 5. Mapping of the n-dimensional algorithm for 4x4 and 7x2 topologies.

3.4. Results

Table 1 compares the results of our algorithm with the Müller and Resch (M&R) one and the T3E default mapping using the average number of hops (d_{av}) as metric. In some cases we only indicate that the algorithm is sub-optimal, so d_{av}>1:

Grid	Non-cyclic			Cyclic		
	MPI	M&R	Greedy	MPI	M&R	Greedy
32x1x1	1.61	>1	1	1.625	>1	1
4x4x1	2	1	1	2.25	1	1
8x4x1	2.2	1	1	2.4	>1	1
2x2x2	1.3	1.3	1	2.0	>1	1
3x3x3	2.27	1.8	1.8	2.34	>1.8	1.9

Table 1. Average number of hops that a message has to travel. MPI and greedy refer to the default mapping and our algorithm respectively.

4. `MPI_Cart_create` Benchmark

Although the average number of hops can be used to estimate the quality of the mapping algorithm, the time required for exchanging data is the only definite metric. Therefore, we have used a synthetic benchmark to simulate the communication pattern that can be found in standard domain decomposition applications:

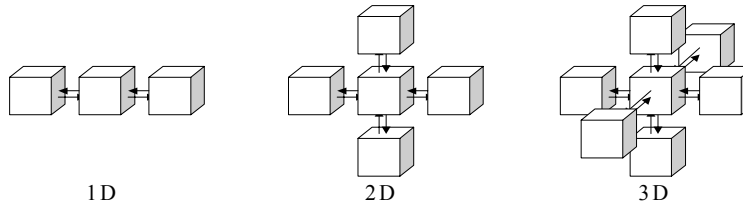


Fig. 6. Communication pattern in standard domain decomposition methods.

Results for communications in buffered mode are shown in the following figure:

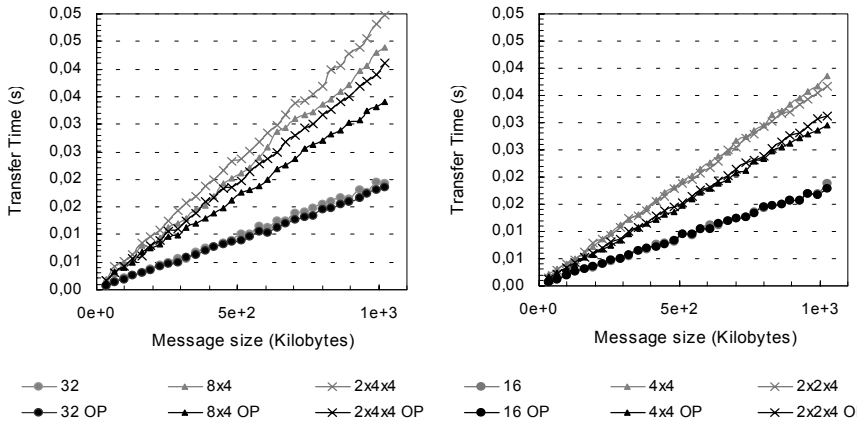


Fig. 7. `MPI_Cart_create` benchmark results for different cartesian topologies, with (**OP**) and without reordering, using 32 PE (left-hand char) and 16 PE (right-hand chart).

Apart from 1D topologies, improvements are always significant. Table 2 presents the *asymptotic bandwidth* obtained with 32 PEs:

Topology	Non-cyclic communication		Cyclic communication	
	Buffered mode	Synchronous mode	Buffered mode	Synchronous mode
1D	219 → 221	300 → 300	219 → 219	300 → 300
2D	213 → 263	262 → 340	141 → 200	181 → 280
3D	219 → 262	267 → 351	194 → 230	255 → 300

Table 2. Improvements in the asymptotic bandwidths (MB/s) for 32 PEs.

For non-cyclic communications in buffered mode, improvements are around 20% for 2D and 3D topologies. As in the random pairwise exchange, performance is better using the synchronous mode. Obviously, the improvement obtained by the optimized mapping is greater in this mode, around 40%, since buffered mode helps to reduce the contention problem.

The behavior for 1D topologies agrees with some previous measurements of the network contention influence on message passing. As we have shown in [1], the impact of contention is only significant when processors involved in a message exchange are 2 or more hops apart. Although the MPI default mapping is sub-optimal for 1D topologies, the network distance between neighbours is almost always lower than 2 in this case, and hence, the optimal mapping improvements are not relevant.

For cyclic communications improvements are even greater than those obtained in the non-cyclic case, reaching around 50% for the 2D topology using synchronous mode.

5. Conclusions

We have first shown how PE mapping affects actual communication bandwidths on the T3E and the Origin 2000. The performance depends on the mapping on both systems, although the influence is more significant on the T3E where the difference between optimal and worst mappings grows with message size.

In view of these results, we conclude that the physical placement must be accounted for in both systems. The SGI Origin 2000 provides for this need through `dplace`. However, a similar tool does not exist in the T3E. For this reason we have proceeded to study a greedy mapping algorithm for n-dimensional cartesian topologies on the T3E, which can be used as an optimization for the `MPI_Cart_create` function. Compared to the MPI default mapping, our algorithm reduces the average number of hops that a message has to travel.

Finally, to measure the influence of the processor mapping on performance, we have used a synthetic benchmark to simulate the communication pattern found in standard domain decomposition applications. The improvements are significant in most cases, reaching around 40% in 2D and 3D topologies with synchronous mode.

Although our experiments have been focused on cartesian topologies, we believe that they open the possibility of optimizations in other topologies (e.g. graphs).

At the time of writing this paper we are applying this optimization to actual applications instead of synthetic benchmarks and we are probing it on a larger T3E system.

Acknowledgements

This work has been supported by the Spanish research grant TIC 1999-0474. We would like to thank Ciemat and CSC (Centro de Supercomputación Complutense) for providing access to their parallel computers.

References

[1] M. Prieto, D. Espadas I. M. Llorente, F. Tirado. “Message Passing Evaluation and Analysis on Cray T3E and Origin 2000 systems”, in Proceedings of Europar 99, pp. 173-182. Toulouse, France, 1999.
[2] M. Prieto, I. M. Llorente, F. Tirado. “Partitioning of Regular Domains on Modern Parallel Computers”, in Proceedings of VECPAR 98, pp. 305-318. Porto, Portugal, 1998.
[3] E. Anderson, J. Brooks, C.Grass, S. Scott. “Performance of the CRAY T3E Multiprocessor”, in Proceedings of SC97, November 1997.
[4] David Culler, Jaswinder Pal Singh, Annop Gupta. "Parallel Computer Architecture. A hardware/software approach" Morgan-Kaufmann Publishers, 1998.
[5] Michael Resch, Holger Berger, Rolf Rabenseifner, Tomas Bönish "Performance of MPI on the CRAY T3E-512", Third European CRAY-SGI MPP Workshop, PARIS (France), Sept. 11 and 12, 1997.
[6] J. Laudon and D. Lenoski. “The SGI Origin: A ccNUMA Highly Scalable Server”, in Proceedings of ISCA’97. May 1997.
[7] Aad J. van der Steen and Ruud van der Pas "A performance analysis of the SGI Origin 2000", in Proceedings of VECPAR 98, pp. 319-332. Porto, Portugal, 1998.
[8] Origin 2000 and Onyx2 Performance Tuning and Optimization Guide. Chapter 8. Available at http://techpubs.sgi.com.
[9] Matthias Müller and Michael M. Resch , "PE mapping and the congestion problem on the T3E" Hermann Lederer and Friedrich Hertweck (Ed.), in Proceedings of the 4th European Cray-SGI MPP Workshop, IPP R/46, Garching/Germany 1998.

Performance Prediction of an NAS Benchmark Program with ChronosMix Environment

Julien Bourgeois and François Spies

LIFC,
Université de Franche-Comté,
16, Route de Gray,
25030 BESANCON Cedex,
FRANCE
{bourgeoi,spies}@lifc.univ-fcomte.fr,
http://lifc.univ-fcomte.fr/

Abstract. The Networks of Workstations (NoW) are becoming real distributed execution platforms for scientific applications. Nevertheless, the heterogeneity of these platforms makes complex the design and the optimization of distributed applications. To overcome this problem, we have developed a performance prediction tool called ChronosMix, which can predict the execution time of a distributed algorithm on parallel or distributed architecture. In this article we present the performance prediction of an NAS Benchmark program with the ChronosMix environment. This study aims at emphasizing the contribution of our environment in the performance prediction process.

1 Introduction

Usually scientific applications are intended to run only on dedicated multiprocessor machines. With the continual increase in workstation computing powers and especially the explosion of communication speed, the networks of workstations (NoW) became possible distributed platforms of execution and inexpensive for scientific applications. Its main problem lies in the heterogeneity of NoW compared to the homogeneity of the multiprocessor machines. In a NoW, it is difficult to allocate the entire work in an optimal manner, it is difficult to know the exact benefit or if there is any or simply to know which best algorithm to solve a specific problem is. Therefore, the optimization of the distributed application is a hard task to achieve.

A way to meet these objectives is to use performance prediction.

We identify three categories of tools that realize performance evaluation of a parallel program. Their objectives are quite different, because they use three different types of input: processor language, high-level language or modeling formalism.

The aim of tools based on a processor language, like Trojan [Par96] and SimOS [RBDH97], is to provide a very good accuracy. Thus, they avoid the introduction of compiler perturbations due to optimization. But, they work on

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 208–216, 2000.
© Springer-Verlag Berlin Heidelberg 2000

unique architecture and cannot be extended to any other. Finally, this tool category implies an important slowdown.

The aim of tools based on high-level language is to allow adapted accuracy in designing parallel applications with a minimum slowdown. Mainly, this tool category has the ability to calculate application performance on various types of architecture from an execution trace. P3T [Fah96], Dimemas [GCL97], Patop [WOKH96] and ChronosMix [BST99] belong to this category. The P3T tool is part of the Vienna Fortran Compilation System and its aim is to evaluate and classify parallel strategies. P3T helps the compiler to find the appropriate automatic parallelization of the sequential Fortran program in homogeneous architecture. The Dimemas tool is part of the Dip environment [LGP+96]. It simulates distributed memory architecture. It exclusively uses trace information and does not realize any program model. So, Dimemas is able to evaluate binaries like commercial tools and various types of architecture and communication libraries like PVM and MPI. Patop is a performance prediction tool based on an on-line monitoring system. It is part of The Tool-set [WL96] and allow to analyze performances on homogeneous multi-computers.

The aim of a modeling formalism tool is to simplify the application description and to put it in a tuning up form. With this type of tools, the accuracy is heavily linked with the modeling quality. Pamela [Gem94] and BSP [Val90] are from this category. The Pamela formalism is strong enough to allow complex system modeling. Various case studies have been conducted with Pamela and many algorithm optimizations have been realized.

However, current tools and methods have disadvantages that do not allow an efficient use of performance prediction. The three main disadvantages of performance evaluation tools are:

- The slowdown is the ratio between the time to access simulation results and real execution time. It is calculated for one processor and the different times must come from the same architecture.
- The use constraints
- The lack of heterogeneous support

Our tool, ChronosMix, has been developed by taking account of these aspects. The slowdown has been minimized, the use has been improved thanks to automatic modeling and the heterogeneity has been integrated. These main aspects are emphasized in a case study.

This paper starts in section 2 with a description of the performance evaluation tool for a parallel program called ChronosMix. It ends in section 3 with a case study from the NAS Benchmark.

2 Presentation of the ChronosMix Environment

The purpose of ChronosMix [BST99] is to provide the most complete performance prediction environment as possible in order to help in the designing and the optimizing of distributed applications. To do so, ChronosMix environment comprises different modules:

- Parallel architecture modeling (MTC and CTC)
- Program modeling (PIC)
- Simulation engine
- Graphical interface (Predict and ParaGraph)
- Database web server

Figure 1 illustrates the relations between these different modules.

Parallel architecture modeling consists of two tools the MTC (Machine Time Characteriser) and the CTC (Communication Time Characteriser). The MTC is a micro-benchmarking tool which measures the execution time of instructions of which a machine instruction set is composed. A given local resource will just be assessed once by the MTC, meaning that the local resource model can be useful in the simulation of all program models. The CTC is an expansion of SKaMPI [RSPM98]. It measures MPI communication times by varying the different parameters. CTC measures are written in a file, making it necessary for a cluster to be measured just once.

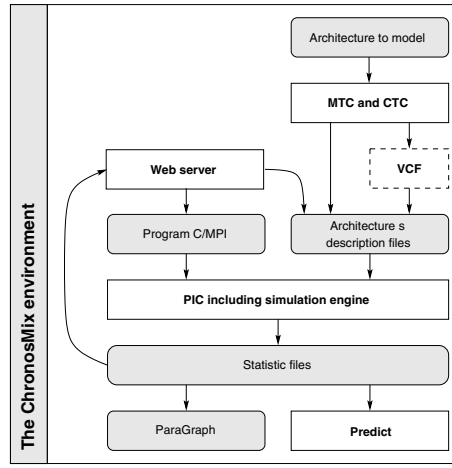


Fig. 1. The ChronosMix environment

The C/MPI program modeling and the simulation engine are both included in the PIC (Program Instruction Characteriser), which is a C++ 15,000-line program. Figure 2 shows how the PIC works. On the left side of figure 2 the method for analyzing the input C/MPI program will be entirely static. Indeed, the execution number of each block and the execution time prediction are calculated statically. On the right, you can see a program passing through the PIC in a semi-static way. The number of executions is attributed to each block by means of an execution and a trace. However, the execution time prediction phase is static, which explains why the program is analyzed through the PIC in a semi-static way.

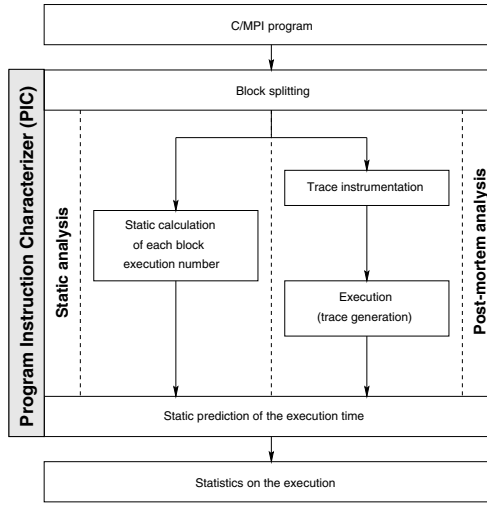


Fig. 2. PIC internal structure

ChronosMix environment currently comprises two graphical interfaces called ParaGraph and Predict. A third interface, the VCF (Virtual Computer Factory), is being developed which will allow new performance prediction architecture to be made.

3 Performance Prediction of the NAS Integer Sorting Benchmark

3.1 Presentation of the Integer Sorting Benchmark

The program used for testing the efficiency of our performance prediction tool is from the NAS (Numerical Aerodynamic Simulation) Parallel Benchmark [BBDS92] developed by the NASA. The NAS Parallel Benchmark (NPB) is a set of parallel programs designed to compare the parallel machine performances according to the criteria belonging to the aerodynamic problem simulation. These programs are widely used in numerical simulations. One of the 8 NPB programs is the parallel Integer Sorting (IS) [Dag91]. This program is based on the barrel-sort method. The parallel integer sorting is particularly used in the Monte-Carlo simulations integrated in the aerodynamic simulation programs.

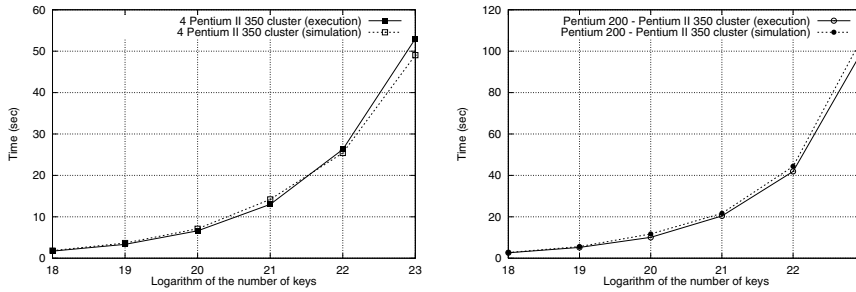
IS consists of various types of MPI communication. Indeed, it uses asynchronous point-to-point communications, broadcasting and gathering functions as well as all-to-all functions. IS therefore deals with network utilization in particular. For that matter, the better part of its execution time is spent in communication.

3.2 Comparison of IS on Various Types of Architecture

Four types of architecture have been used to test the ChronosMix accuracy :

- A cluster composed of 4 Pentium II 350Mhz with Linux 2.0.34, gcc 2.7.2.3 and a 100 Mbit/s Ethernet commuted network.
- A heterogeneous cluster composed of 2 Pentium MMX 200Mhz and 2 Pentium II 350Mhz with Linux 2.0.34, gcc 2.7.2.3 and a 100 Mbit/s Ethernet commuted network.
- A cluster composed of 4 DEC AlphaStation 433Mhz with Digital UNIX V4.0D, the DEC C V5.6-071 compiler and a 100 Mbit/s Ethernet commuted network.
- The same cluster but with 8 DEC AlphaStation 433Mhz

All the execution times are an average of 20 executions. Figure 3 shows the IS execution and the IS prediction on the Pentium clusters. The error percentage between prediction and real execution is represented in figure 5. The cluster comprising only Pentium II is approximately twice as fast as the heterogeneous cluster. Replacing the two Pentium 200 with two Pentium II 350 is interesting - the error noticed between the execution time and the prediction time remains acceptable since it is not over 15% for the heterogeneous cluster and 10% for the homogeneous cluster, the average error amounting to 7% for the heterogeneous and the homogeneous cluster.



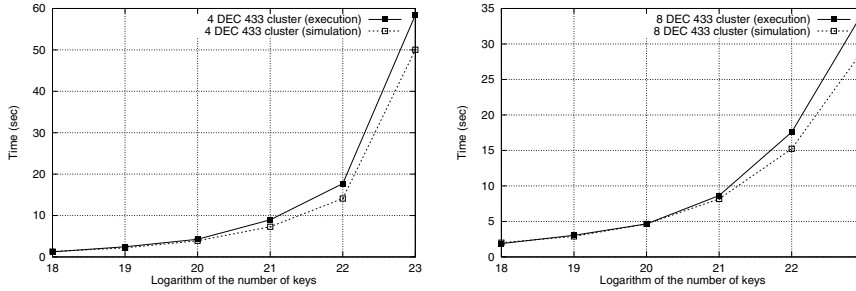
(a) IS on a 4 Pentium II cluster

(b) IS on a 2 Pentium 200 - 2 Pentium II cluster

Fig. 3. Execution and prediction of IS on Pentium clusters

Figures 4(a) and 4(b) present IS execution and IS prediction respectively for the 4 DEC AlphaStation cluster and for the 8 DEC AlphaStation cluster. In parallel with figure 5 they show that the difference between prediction and execution remain acceptable. Indeed, concerning the 4 DEC AlphaStation cluster, the difference between execution and prediction is not over 20% and the average

is 12%. As for the 8 DEC AlphaStation cluster, the maximum error is 24% and the average is 9%.



(a) 4 DEC AlphaStation cluster (b) 8 DEC AlphaStation cluster

Fig. 4. IS on 4 and 8 DEC AlphaStation clusters

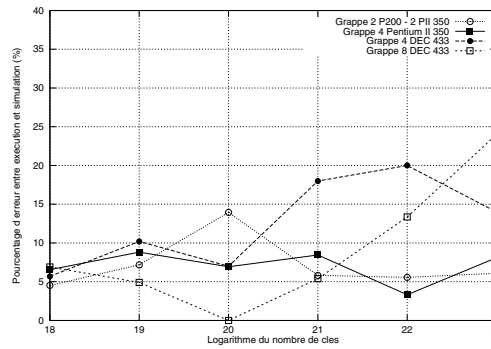
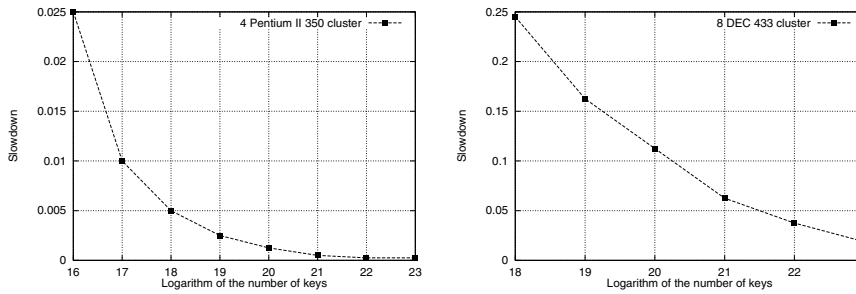


Fig. 5. Error percentage between prediction and execution on the three type of architecture

The integer sorting algorithm implemented by IS is frequently found in numerous programs and the executions and the simulations have dealt with real-size problems. Therefore, on a useful, real-size program, ChronosMix proved able to give relatively accurate results.

The other quality of ChronosMix lies in how quickly the results are obtained. Indeed, if a simulator is too long in giving the results, the latter can become quite useless. Even in semi-static mode, ChronosMix can prove to be faster than an execution.

A trace is generated for a given size of cluster and for a given number of keys. This trace, generated on any computer capable of running an MPI program. Concretely, traces have been generated on a bi-Pentium II 450 and were then casually used in the IS prediction on the 4 DEC AlphaStation cluster and on the 4 Pentium II cluster. This shows it is difficult to count the trace generating time in the total performance prediction time. Figures 6(a) and 6(b) show the IS performance prediction slowdown. Normally, the slowdown of a performance prediction tool is greater than 1, meaning that the performance prediction process is longer than the execution. For this example, the ChronosMix slowdown is strictly smaller than 1, meaning that by simulated processor, performance prediction is faster than the execution. This result shows that slowdown is an inadequate notion for ChronosMix. Performance prediction of the Pentium II cluster is at least 10 times faster than the execution. With the maximum number of keys, ChronosMix is 1000 times faster in giving the program performances than the real execution. Concerning the 8 DEC AlphaStation cluster, the ChronosMix slowdown is always below 0.25 and falls to 0.02 for a number of keys of 2^{23}. If the slowdown is reduced when the number of keys rises, it is simply because the performance prediction time is constant whereas the execution time rises sharply according to the number of keys to sort.



(a) 4 Pentium II cluster (b) 8 DEC AlphaStation cluster

Fig. 6. Slowdown for the IS performance prediction

4 Conclusion

The IS performance prediction has revealed three of the main ChronosMix qualities:

Its speed. On average, ChronosMix is much faster in giving the application performances than an execution on the real architecture.

Its accuracy. On average, the difference between the ChronosMix prediction and the execution on the different types of architecture is 10%.

Its adaptability ChronosMix proved it could adapt to 2 types of parallel architecture and to several sizes of cluster.

The ability to model distributed system architecture with a set of micro-benchmarks allows ChronosMix to take heterogeneous architecture completely into account. Indeed, in a sense, modeling is automatic, because simulation parameters are assigned by the MTC execution to the local resources and by the CTC execution between workstations. The distributed architecture is simply and rapidly modeled, which allows to follow the processor evolution, but also to adapt our tool to a wide range of architecture.

It is possible to build target architecture from existing one by extending the distributed architecture, e.g. a set of one thousand workstations. It is also possible to modify all the parameters of the modeled architecture, e.g. to stretch the network bandwidth or to increase the floating-point unit power four-fold.

References

[BBDS92] D.H. Bailey, E. Barszcz, L. Dagum, and H. Simon. The NAS parallel benchmarks results. In *Supercomputing'92*, Minneapolis, November 16–20 1992.

[BST99] J. Bourgeois, F. Spies, and M. Trhel. Performance prediction of distributed applications running on network of workstations. In H.R. Arabnia, editor, *Proc. of PDPTA'99*, volume 2, pages 672–678. CSREA Press, June 1999.

[Dag91] L. Dagum. Parallel integer sorting with medium and fine-scale parallelism. Technical Report RNR-91-013, NASA Ames Research Center, Moffett Field, CA 94035, 1991.

[Fah96] T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* ISBN 0-7923-9708-8. Kluwer Academic Publishers, Boston, USA, March 1996.

[GCL97] S. Girona, T. Cortes, and J. Labarta. Analyzing scheduling policies using DIMEMAS. *Parallel Computing*, 23(1-2):23–24, April 1997.

[Gem94] A.J.C. van Gemund. The PAMELA approach to performance modeling of parallel and distributed systems. In editors G.R. Joubert et al., editor, *Parallel Computing: Trends and Applications*, pages 421–428. 1994.

[LGP+96] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A parallel program development environment. In *Proc. of Euro-Par'96*, number 19 in 2, pages 665–674, Lyon, France, August 1996.

[Par96] D. Park. *Adaptive Execution: Improving performance through the runtime adaptation of performance parameters.* PhD thesis, University of Southern California, May 1996.

[RBDH97] M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM TOMACS Special Issue on Computer Simulation*, 1997.

[RSPM98] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. *LNCS*, 1497:52–59, 1998.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM (CACM)*, 33(8):103–111, August 1990.

[WL96] R. Wismüller and T. Ludwig. The Tool-set – An Integrated Tool Environment for PVM. In H. Lidell and al., editors, *Proc. HPCN*, pages 1029–1030. Springer Verlag, April 1996.

[WOKH96] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, March 1996.

Topic 03
Scheduling and Load Balancing

Bettina Schnor

Local Chair

Scheduling and load balancing techniques are key issues for the performance of parallel applications. However, a lot of problems regarding, for example, dynamic load balancing are still not sufficiently solved.

Hence, many research groups are working on this field, and we are glad that this topic presents several excellent results. We want to mention only a few ones from the 13 papers (1 distinguished, 8 regular, and 4 short papers), selected from 28 submissions.

One session is dedicated to system level support for scheduling and load balancing. In their paper *The Impact of Migration on Parallel Job Scheduling for Distributed Systems*, Zhang, Franke, Moreira, and Sivasubramaniam show how back-filling gang scheduling may profit from an additional migration facility. Leinberger, Karypis, and Kumar present in *Memory Management Techniques for Gang Scheduling* a new gang scheduling algorithm which balances the workload not only due to processor load, but also due to memory utilization.

The wide range of research interests covered by the contributions to this topic is illustrated by two other interesting papers. *Load Scheduling Using Performance Counters* by Lindenmaier, McKinley, and Temam presents an approach for extracting fine-grain run-time information for hardware counters to improve instruction scheduling. Gursoy and Atun investigate in *Neighbourhood Preserving Load Balancing: A Self-Organizing Approach* how Kohonen's self-organizing maps can be used for static load balancing.

Further, there are several papers dealing with application-level scheduling in Topic 3. One of these, *Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning* by Schloegel, Karypis, and Kumar was selected as distinguished paper. It investigates the load balancing requirements of multi-phase simulations.

We would like to thank sincerely the more than 40 referees that assisted us in the reviewing process.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 217–217, 2000.
© Springer-Verlag Berlin Heidelberg 2000

A Hierarchical Approach to Irregular Problems*

Fabrizio Baiardi, Primo Becuzzi, Sarah Chiti, Paolo Mori, and Laura Ricci

Dipartimento di Informatica, Universitá di Pisa
Corso Italia 40, 50125 - PISA
<last name>@di.unipi.it

Abstract. Irregular problems require the computation of some properties for a set of elements irregularly distributed in a domain in a dynamic way. Most irregular problems satisfy a locality property because the properties of an element e depend upon the elements "close" to e. We propose a methodology to develop a highly parallel solution based on load balancing strategies that respects locality, i.e. e and most of the elements close to e are mapped onto the same processing node. We present the experimental results of the application of the methodology to the n-boby problem and to the adaptive multigrid method.

1 Introduction

The solution of an irregular problem requires the computation of some properties for each of a set of elements that are distributed in a n-dimensional domain in an irregular way, that changes during the computation. Most irregular problems satisfy a locality property because the probability that the properties of an element e_i affects those of e_j decreases with the distance from e_i to e_j. Examples of irregular problems are the Barnes-Hut method [2], the adaptive multigrid method [3] and the hierarchical radiosity method [5].

This paper proposes a parallelization methodology for irregular problems in the case of distributed memory architectures with a sparse interconnection network. The methodology defines two load balancing strategies to, respectively, map the elements onto the processing nodes, p-nodes, and update the mapping as the distribution changes and a further strategy to collect information on elements mapped onto other p-nodes. To evaluate its generality, the methodology has been applied to the Barnes-Hut method for the n-body problem, NBP, and to the adaptive multigrid method, AMM. Sect. 2 describes the representation of the domain and the load balancing strategies and Sect. 3 presents the strategy to collect remote data. Experimental results are discussed in Sect. 4.

2 Data Mapping and Runtime Load Balancing

All the strategies in our methodology are defined in terms of a hierarchical representation of the domain and of the element distribution. At each hierarchical

* This work was partially supported by CINECA

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 218–222, 2000.
© Springer-Verlag Berlin Heidelberg 2000

level, the domain is partitioned into a set of equal subdomains, or spaces. The hierarchy is described through the **Hierarchical Tree**, *H-Tree* [7, 8]; the root represents the whole domain, each other node N, **hnode**, represents a space, *space(N)*, and it records information on the elements in *space(N)*. A space A that violates a problem dependent condition, is partitioned into 2^n equal subspaces by halving each of its sides. A is partitioned if contains more than one body in the NBP, and if the current approximation error in its vertexes is larger than a threshold in AMM. The sons of N describe the partitioning of *space(N)*. In the following, *hnode(A)* denotes the hnode representing the space A, and the level of A is the depth of *hnode(A)* in the H-Tree. Hnodes representing larger spaces record a less detailed information than those representing smaller spaces. In the NBP, each leaf L records the mass, the position in the space and the speed vector of the body in *space(L)*, while any other hnode N records the center of gravity and the total mass of the bodies in *space(N)*. In the AMM, each hnode N records the coordinates, the approximated solution of the differential equation and the evaluation of the error of the point on the leftmost upward vertex of *space(N)*. At run time, the hierarchy and the H-Tree are updated according to the current elements distribution. Since the H-Tree is too large to be replicated in each p-node, we consider a subset that is replicated in each p-node, the RH-Tree, and one further subset, the private H-Tree, for each p-node.

To take locality into account, we define the initial mapping in three steps: spaces ordering, workload determination and spaces mapping onto p-nodes.

The spaces are ordered through a space filling curve *sf* built on the spaces hierarchy [6]; *sf* also defines a visit *v(sf)* of the H-Tree that returns a sequence $S(v(sf)) = [N_0, .., N_m]$ of hnodes. The load of a hnode N evaluates the amount of computations due to the elements in *space(N)*. In the NBP, the load of a leaf L is due to the computation of the force on the body in *space(L)*. This load is distinct for each leaf and it is measured during the computation, because it depends upon the current body distribution. No load is assigned to the other hnodes because no forces are computed on them. Since in the AMM the same computation is executed on each space, the same load is assigned to each hnode.

The *np* p-nodes are ordered in a sequence $SP = [P_0, .., P_{np}]$ such that the cost of an interaction between P_i and P_{i+1} is not larger than the cost of the same interaction between P_i and any other p-node. Since each p-node executes one process, P_k denotes also the process executed on the k-th p-node of SP.

S(v(sf)) is partitioned into *np* segments, whose overall load is as close as possible to *average_load*, the ratio between the overall load and *np*. We cannot assume that the load of each segment S is equal to *average_load* because each hnode is assigned to one segment; in the following, $= (S, C)$ denotes that the load of S is as close as possible to C. The first segment of $S(v(sf))$ is mapped onto P_0, the second onto P_1 and so on. This mapping satisfies the **range property**: *if the hnodes N_i and N_{i+j} are assigned to P_h, then all the hnodes in-between N_i and N_{i+j} in $S(v(sf))$, are assigned to P_h as well.* Due to the property of space filling curves, any mapping satifying this property allocates elements that are

close to each other to the same p-node. Furthermore, two consecutive segments are mapped onto p-nodes that are close in the interconnection network.

PH-Tree(P_h), the private H-Tree of P_h, describes Do_h, the segment assigned to P_h, and includes a hnode N if *space(N)* belongs to Do_h. The RH-Tree is the union of the paths from the H-Tree root to the root of each private H-Tree; each hnode N records the position of *space(N)* and the owner process. In the NBP, a hnode N belongs to PH-Tree(P_h) iff all the leaves in *Sub(N)*, the subtree rooted in N, belong to this tree too, otherwise it belongs to the RH-Tree. To minimize the replicated data, the intersection among a private H-Tree and the RH-Tree includes the roots of the private H-Trees only. In the AMM, each hnode belongs to the private H-Tree of a p-node, because all hnodes are paired with a load.

Due to the body evolution in the NBP and to the grid refinement in the AMM, the initial allocation could result in an unbalance at a later iteration. The mapping is updated if the largest difference between *average_load* and the current workload of a process is larger than a tolerance threshold $T > 0$. Let us suppose that the load of P_h is *average_load* + C, $C > T$, while that of P_k, $h \neq k$, is *average_load* - C. To preserve the range property, the spaces are shifted among all the processes P_i in-between P_h and P_k. Let us define $Prec_i$ as the set $[P_0...P_{i-1}]$ and $Succ_i$ as the set $[P_{i+1}...P_{np}]$. Furthermore, $Sbil(PS)$ is the global load unbalances of the set PS. If $Sbil(Prec_i) = C > T$, i.e. processes in $Prec_i$ are overloaded, P_i receives from P_{i-1} a segment S where $= (S, C)$. If, instead, $Sbil(Prec_i) = C < -T$, P_i sends to P_{i-1} a segment S where $= (S, C)$. The same procedure is applied to $Sbil(Succ_i)$, but the hnodes are either sent to or received from P_{i+1}. To preserve the range property, if $Do_i = [N_q....N_r]$, then P_i sends to P_{i-1} a segment $[N_q....N_s]$, while it sends to P_{i+1} a segment $[N_t....N_r]$, with $q \leq t, s \leq r$.

3 Fault Prevention

To allow P_h to compute the properties of elements in Do_h whose neighbors have been mapped onto other p-nodes, we have defined the **fault prevention** strategy. The fault prevention strategy allows P_h to receive the properties of the neighbors of elements in Do_h without requesting them. Besides reducing the number of communications, this simplifies the applications of some optimization strategies such as messages merging. For each space A in Do_k, P_k determines, through the neighborhood stencil, which processes require the data of A and sends to these processes the data, without any explicit request. To determine the data needed by P_h, P_k exploits the information on Do_h in the RH-Tree. In general, P_k approximates these data because the RH-Tree records a partial information only. The approximation is always *safe*, i.e. it includes any data P_h needs, but, if it is not accurate, most data is useless. To improve the approximation, the processes may exchange some information about their private H-Trees before the fault prevention phase (**informed fault prevention**).

In the NBP, the neighborhood stencil of a body b is defined by the "Multipole Acceptability Criterium" (MAC), that determines, for each hnode N, whether

the interaction between b and the bodies in *space(N)* can be approximated. A widely adopted definition of the MAC [2] is $\frac{l}{d} < \theta$, where l is the length of the side of *space(N)*, d is the distance between b and the center of gravity of the bodies in *space(N)* and θ is an user defined approximation coefficient. P_k computes the *influence space*, *is(N)*, for each hnode N that is not a leaf of PH-Tree(P_k). *is(N)* is a sphere with radius $\frac{l}{\theta}$ centered in the center of gravity recorded in N. Then, P_k visits PH-Tree(P_k) in anticipated way and, for each hnode N that is not a leaf, it computes $J(N, R) = is(N) \cap space(R)$ where R is the root of PH-Tree(P_h), $\forall h \neq k$. If $J(N, R) \neq \emptyset$, it may include one body d, and the approximation cannot be applied by P_h when computing the forces on d. Hence, P_h needs the information recorded in the sons of N in the PH-Tree(P_k). To guarantee the safeness of fault prevention, P_k assumes that $J(N, R)$ always includes a body, and it sends to P_h the sons of N. P_h uses these data iff $J(N, R)$ includes at least one body. If $J(N, R) = \emptyset$ then, for each body in Do_h, P_h approximates the interaction with N and it does not need the hnodes in *Sub(N)*.

In the AMM, P_h applies the multigrid operators, in the order stated by the V-cycle, to the points in Do_h [3, 4]. We denote by Bo_h the boundary of Do_h, i.e. the sets of the spaces in Do_h such that one of their neighbors does not belong to Do_h. Bo_h depends upon the neighborhood stencil of the operator *op* that is considered. Let us define $I_{h,op,liv}$ as the set of spaces not belonging to Do_h and including the points required by P_h to apply *op* to the points in the spaces at level *liv* of Bo_h. $\forall h \neq k$, P_k exploits the information in the RH-Tree about Do_h to determine the spaces in Do_k that belongs to $I_{h,op,liv}$. Hence, it computes and sends to P_h a set $A_k I_{h,op,liv}$ that approximates $I_{h,op,liv} \cap Do_k$. The values of points in $A_k I_{h,op,liv}$ are trasmitted just before the application of *op*, because they are updated by the previous operators in the V-cycle. To improve the approximation, we adopt *informed fault prevention*. If a space in Do_k belongs to $I_{h,op,liv}$, $k \neq h$, P_h sends to P_k, at the beginning of the V-cycle and before the fault prevention phase, the level of each space in Bo_h that could share a side with the one in Do_k. If the load balancing procedure has been applied, P_h sends the level of all the spaces in Bo_h, otherwise, since spaces are never pruned, P_h sends the level of the new spaces only.

4 Experimental Results

To evaluate the generality of our methodology, we have implemented the NBP on the Meiko CS 1 with OCCAM II as programming language and the AMM on a Cray T3E with C and MPI primitives. The data set for the NBP is generated according to [1]. The AMM solves the *Poisson's problem* in two dimensions subject to two different boundary conditions, denoted by $h1$ and $h2$:

$$h1(x, y) = 10 \qquad h2(x, y) = 10\cos(2\pi(x - y))\frac{\sinh(2\pi(x + y + 2))}{\sinh(8\pi)}$$

To evaluate the fault prevention strategy, we consider the ratio of the amount of data sent against those that are really needed. This ratio is less than 1.1 in the

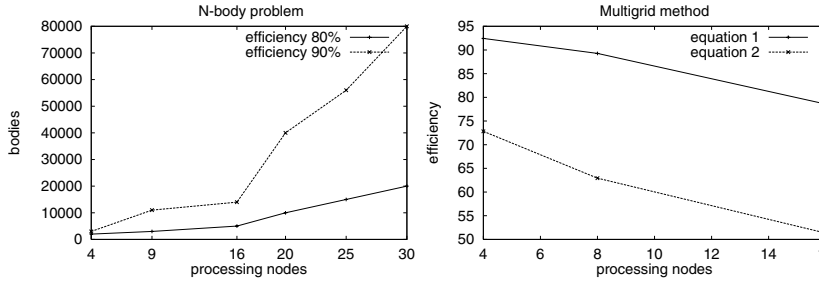


Fig. 1. Efficiency

NBP and less than 1.24 in the AMM. In the AMM, informed fault prevention reduces the ratio to 1.04.

In both problems, the balancing procedure reduces the total execution time but the optimal value of T has to be determined. In the NBP, the execution time is nearly proportional to difference between the adopted value of T and the optimal one. In the AMM, the optimal value of T also depends upon the considered equation, that determines the structure of the H-Tree. In this case, the relative difference between the execution time of a well balanced execution and that of an unbalanced one can be larger than 25%.

Fig. 1 shows the efficiency of the two implementations. For the NBP, the lowest number of bodies to achieve a given efficiency is shown. For the AMM we show the results for the two equations, for a fixed number of initial points, 16.000, and the same maximum depth of the H-Tree, 12. The larger granularity of the NBP results in a better efficiency. In fact, after each fault prevention phase, the computation is executed on the whole private H-Tree in the NBP while in AMM it is executed on one level of this tree.

References

[1] S. J. Aarset, M. Henon, and R. Wielen. Numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37(2), 1974.

[2] J.E. Barnes and P. Hut. A hierarchical O(nlogn) force calculation algorithm. *Nature*, 324, 1986.

[3] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Physics*, 53, 1984.

[4] W. Briggs. *A multigrid tutorial.* SIAM, 1987.

[5] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4), 1991.

[6] J.R. Pilkington and S.B. Baden. Dynamic partitioning of non–uniform structured workloads with space filling curves. *IEEE TOPDS*, 7(3), 1996.

[7] J.K. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, California Institute of Technology, 1990.

[8] J.P. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors.* PhD thesis, Stanford University, 1993.

Load Scheduling with Profile Information*

Götz Lindenmaier[1], Kathryn S. McKinley[2], and Olivier Temam[3]

[1] Fakultät für Informatik, Universität Karlsruhe
[2] Department of Computer Science, University of Massachusetts
[3] Laboratoire de recherche en informatique, Universite de Paris Sud

Abstract. Within the past five years, many manufactures have added hardware performance counters to their microprocessors to generate profile data cheaply. We show how to use Compaq's DCPI tool to determine load latencies which are at a fine, instruction granularity and use them as fodder for improving instruction scheduling. We validate our heuristic for using DCPI latency data to classify loads as hits and misses against simulation numbers. We map our classification into the Multiflow compiler's intermediate representation, and use a locality sensitive Balanced scheduling algorithm. Our experiments illustrate that our algorithm improves run times by 1% on average, but up to 10% on a Compaq Alpha.

1 Introduction

This paper explores how to use hardware performance counters to produce fine grain latency information to improve compiler scheduling. We use this information to hide latencies with any available instruction level parallelism (ILP). (ILP for an instruction is the number of other instructions available to hide its latency, and the ILP of a block or program is the average of the ILP of its instructions). We use DCPI, the performance counters on the Alpha, and Compaq's `dcpicalc` tool for translating DCPI's statistics into a usable form. DCPI provides a very low cost way to collect profiling information, especially as compared with simulation, but it is not as accurate. For instance, `dcpicalc` often cannot produce the reason for a load stall. We show nevertheless it is possible to attain fine grain latency information from performance counters. We use a heuristic to classify loads as hits and misses, and this classification matches simulation numbers well. We are the first to use performance counters at such a fine granularity to improve optimization decisions.

In the following, Section 2 presents related work. Section 3 describes DCPI, the information it provides, how we can use it, and how our heuristic compares with simulation numbers. Section 4 briefly describes the load sensitive scheduling algorithm we use, and how we map the information back to the IR of the compiler. Section 5 presents the results of our experiments on Alpha 21064 and 21164 that show execution time improvements are possible, but our average improvements are less than 1%. Our approach is promising, but it needs new scheduling algorithms that take into account variable latencies and issue width to be fully realized.

* This work is supported by EU Project 28198; NSF grants EIA-9726401, CDA-9502639, and a CAREER Award CCR-9624209; Darpa grant 5-21425; Compaq and by LTR Esprit project 24942 MHAOTEU. Any opinions, findings, or conclusions expressed are the authors' and not necessarily the sponsors'.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 223–233, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Related Work

The related work for this paper falls into three categories: performance counters and their use, latency tolerance, and scheduling. Our contribution is to show how to use performance counters at a fine granularity, rather than aggregate information, and how to tolerate latency by improving scheduling decisions.

We use the hardware performance counters and monitoring on a 4-way issue Alpha [3, 5]. Similar hardware now exists on the Itanium, Intel PentiumPro, Sun Sparc, SGI R10K and in Shrimp, a shared-memory parallel machine [3, 9]. The main advantages of using performance counters instead of software simulation or profiling are time and automation. Performance counters yield information at a cost of approximately 1-2% of execution time, and do not require users to compile with and without profiling. Continuous profiling enables recompilation after program execution to be completely hidden from the user with later, free cycles (our system does not automate this feature). Previous work using performance counters as a source of profile information have used aggregate information, such as the miss rate of a subroutine or basic block [1] and critical path profiles to sharpen constant propagation [2]. Our work is unique in that it uses information at the instruction level, and integrates it into a scheduler.

Previous work on using instruction level parallelism (ILP) to hide latencies for non-blocking caches has two major differences from this work [4, 6, 8, 10, 12]. First, previous work uses static locality analysis which works very well for regular array accesses. Secondly, these schedulers only differentiates between a hit or a miss. Since we use performance counters, we can improve the schedules of pointer based codes that compilers have difficulty analyzing. In addition, we obtain and use variable latencies which further differentiates misses and enables us to concentrate ILP on the misses with the longest observed average latencies.

3 DCPI

This section describes DCPI, `dcpicalc` (a Compaq tool that translates DCPI output to a useful form), and compares DCPI results to simulation. DCPI is a runtime monitoring tool that cheaply collects information by sampling hardware counters [3]. It saves the collected data efficiently in a database, and runs continuously with the operating system. Since DCPI uses sampling, it delivers profile information for the most frequently executed instructions, which are, of course, the most interesting with respect to optimization.

3.1 Information Supplied by DCPI

During monitoring, the DCPI hardware counter tracks the occurrence of a specified event. When the counter overflows, it triggers an interrupt. The interrupt handler saves the program counter for the instruction at the head of the issue queue. `Dcpicalc` interprets the sampled data off-line to provide detailed information about, for example, how often, long, and why instructions stall. If DCPI samples an instruction often, the instruction spends a lot of time at the head of the issue queue, which means that it

suffers long or frequent stalls. `Dcpicalc` combines a static analysis of the binary with the dynamic DCPI information to determine the reason(s) for some stalls. It assumes all possible reasons for stalls it cannot analyze.

```
     instruction          static  dynamic
                          stalls  stalls

1  ldl   r24, -32720(gp)   1      1.0cy
2  lda   r2,  -32592(gp)   0
       i
3  ldl   r26, -32668(gp)   1      1.5cy
4  lda   gp,  0(sp)        0
     b
     b
       d
       d
       d
5  cmplt zero, r26, r26    2      3.5cy
6  ldl   r25, -32676(gp)   0
     b
     b
       i    ...20.0cy
       i
7  cmplt r26, r25, r25     2     20.0cy
     b
8  bis   r24, r25, r25     1      1.0cy
     a
9  beq   r25, 0x800f00     1      1.0cy
```

Fig. 1. Example for the calculation of locality data.

The reasons for a stall are `a`, `b`, `i`, and `d`; `a` and `b` indicate stalls due to an unresolved data dependence on the first or second operand respectively; `i` indicates an instruction cache miss; and `d` indicates a data cache miss. Figure 1 shows an example basic block from `compress`, a SPEC'95 benchmark executed on an Alpha 21164 annotated by `dcpicalc`. Five instructions stall; each line without an instruction indicates a half cycle stall before the next instruction can issue.[1] `Dcpicalc` determines reasons and lengths of `a` and `b` from the static known machine implementation, and `i` and `d` from the dynamic information. For example, instruction 3 stalls one cycle because it waits for the integer pipeline to become available and on average an additional half cycle due to an instruction cache miss. The average stall is very short, which also implies it stalls infrequently. Instruction 7 stalls due to an instruction cache miss, on average 20 cycles.

3.2 Deriving Locality Information

This section shows how to translate the average load latencies from `dcpicalc` into hits and misses for use in our scheduler. We derive the following six values about loads from the `dcpicalc` output. Some are determined (d)ynamically, others are based on (s)tatic features of the program.

- *MissMarked* (d): `dcpicalc` detects a cache miss for this load, i.e., the instruction that uses this load stalls and is marked with `d`.

[1] Because more than two instructions rarely issue in parallel, the output format ignores this case.

- *Stall* (d): the length of a *MissMarked* stall.
- *StatDist* (s): The distance in static cycles between the load and the depending instruction.
- *DynDist* (d): The distance in dynamic cycles between the load and the depending instruction.
- *TwoLoads* (s): The instruction using the data produced by a load marked with *TwoLoads* depends on two loads and it is not clear which one caused the stall.
- *OtherDynStalls* (d): The number of other dynamic stalls between the load and the depending instruction.

For example, instruction 1 in Figure 1 has *MissMarked* = *false*, *Stall* = 0, *StatDist* = 6, *DynDist* = 26.0, *TwoLoads* = *false*, and *OtherDynStalls* = 3. Using these numbers, we reason about the probability of a load hitting or missing, and its average dynamic latency as follows. If a load is *MissMarked*, it obviously misses in the cache on some executions. But *MissMarked* gives no information about how often it misses. *Stall* is long if either the cache misses of this load are long, or if they are frequent. Thus, if a load is *MissMarked* and *Stall* and *StatDist* are large, the probability of a miss is high.

Even when a load misses, it may not stall (*MissMarked* = *false*, *Stall* = *0*) because static or dynamic events may hide its latency. If *StatDist* is larger than the latency of the cache, it will not stall. If *StatDist* does not hide a cache latency, a dynamic stall could, in which case *DynDist* or *OtherDynStalls* are high. The DCPI information is easier to evaluate if *StatDist* is small and thus dynamic latencies are exposed, i.e., the loads are scheduled right before a dependent instruction. We generate the initial binaries assuming a load latency of 1, to expose stalls by cache misses.

The Balanced scheduler differentiates *hits* and *misses*, and tries to put available ILP after *misses* to hide a given fixed miss latency. Although we have the actual expected, dynamic latency, the scheduler we modified cannot use it. Since the scheduler assumes misses by default, we classify a load as a *hit* as follows:

$$\neg MissMarked \;\wedge\; (StatDist < 10) \;\wedge\; (Stall = 0 \vee DynDist < 20)$$

and call it the *strict* heuristic because it conservatively classifies a load as a *hit* only if it did not cause a stall due to a cache miss, and for which it is unlikely that the latency of a cache miss is hidden by the current schedule. It also classifies infrequently missing loads as *misses*. We examined several other heuristics, but none perform as well as the *strict* heuristic. For example, we call the following the *generous* heuristic:

$$(StatDist < 5 \;\wedge\; Stall < 10).$$

As we show below, it correctly classifies more loads as *hits* than the strict heuristic, but it also missclassifies many missing loads as *hits*.

3.3 Validation of the Locality Information

We validated the heuristics by comparing their performance to that of a simulation using a version of ATOM [13] that we modified to compute precise hit rates in the three cache levels of the Alpha 21164. The 21164 has a split first level instruction and data cache. The data cache is a 8 KB direct mapped cache, a unified 96 KB three-way associative

second level on chip cache, and a 4MB third level off chip cache. The latencies of the 21164's first and second cache are 2 and 8 cycles, respectively. (The first level data cache of the 21064 which we use later in the paper has a latency of 3 cycles and is also 8 KB, and the second level cache has a latency of at least 10 cycles.)

Figure 2 summarizes all analyzed loads in eleven SPEC'95[2] and the Livermore benchmarks. The first chart in Figure 2 gives the raw number of loads that hit in the first level cache according to the simulator as a function of how often they hit; each bar represents the number of loads that hit *x%* to *x+10%* in the first level cache. We further divide the 90-100% column into two columns: 90-95% and 95-100% in both figures. Clearly, most loads hit 95-100% of the time.

The second chart in Figure 2 compares how well our heuristics find hits as compared to the simulator. The x-axis is the same as Figure 2. Each bar is the fraction of these loads that the heuristics actually classifies as a hit. Ideally, the heuristics would classify as *hits* all of the loads that hit more than 80%, and none that hit less than 50% of the time. However, since the conservative assumption for our scheduler is miss, we need a heuristic that does not classify loads that mostly miss as *hits*. The generous heuristic finds too many *hits* in loads that usually miss. The strict heuristic instead errs in the conservative direction: it classifies as *hits* only about 40% of the loads that in simulation hit 95% of the time, but it is mistaken less than 5% of the time for those loads that hit less than 50% of the time. In absolute terms these loads are less than 1% of all loads.

4 Scheduling with Runtime Data

In this section, we show how to drive load sensitive scheduling with runtime data. Scheduling can hide the latency of a missing load by placing other useful, independent operations in its delay slots (*behind* it) in the schedule. In most programs, there is not enough ILP to assume all loads are misses in the cache and with the issue width of current processors increasing this problem is exacerbated. With locality information, the scheduler can instead concentrate available ILP behind the missing loads. The ideal

[2] applu, apsi, fpppp, hydro2d, mgrid, su2cor, swim, tomcatv, turb3d, wave5, and compress95.

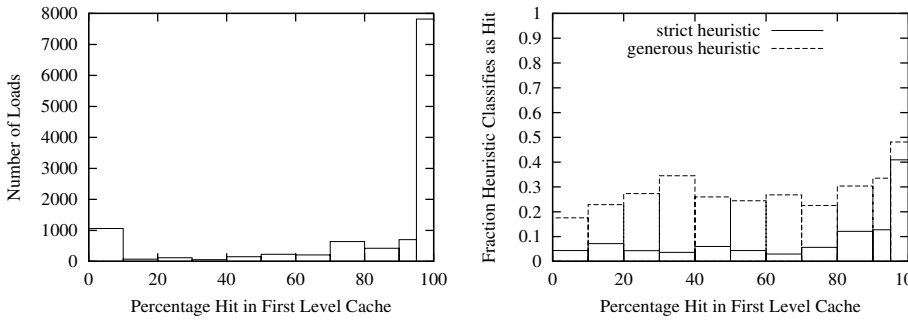


Fig. 2. Simulated number of loads and comparison of heuristics to simulation.

scheduler could differentiate between the expected latency of a miss, placing the most ILP behind the misses with the longest latencies.

4.1 Balanced Scheduling

We use the Multiflow compiler [7, 11] with the Balanced Scheduling algorithm [8, 10], and additional optimizations, e.g., unrolling, to generate ILP and *traces* of instructions that combine basic blocks. Below we first briefly describe Balanced scheduling and then we present our modifications.

Balanced scheduling first creates an acyclic scheduling data dependency graph (DAG) which represents the dependences between instructions. By default it assumes all loads are misses. It then assigns each node (instruction) a weight which is a function of the static latency of the instruction and the available ILP (i.e., how many other instructions may issue in parallel with it).[3] For each instruction `i`, the scheduler finds all others that `i` may come after in the schedule; `i` is thus available as ILP to these other instructions. The scheduler then increases the weight of each instruction after which `i` can execute and hide latency. The usual list scheduling algorithm which tries to cover all the weights then uses this new DAG [8], where the weights reflect a combination of the latency of the instruction and the number of instructions available to schedule with it.

Furthermore, the Balanced scheduler deals with variable load latencies as follows. It makes two passes. The first pass assigns ILP to hide the static latency of all non-load instructions. (For example, the latency of the floating point multiply is known statically and occurs on every execution.) If an instruction has sufficient weight to cover its static latency, the scheduler does not give it any additional weight. In a second pass, the scheduler considers the loads, assigning them any remaining ILP. This structure guarantees that the scheduler first spreads ILP weight to instructions with known static latencies that occur every time the instruction executes. It then distributes ILP weight equally to load instructions which might have additional dynamic latencies due to cache misses. The scheduler thus balances ILP weight across all loads, treating loads uniformly based on the assumption that they all have the same probability of missing in the cache.

4.2 Balanced Scheduling with Locality Data

The Balanced scheduler can further distinguish loads as *hits* or a *misses*, and distribute ILP only to missing loads. The scheduler gives ILP weight only to *misses* after covering all static cycles of non-loads. If ILP is available, the *misses* will receive more weight than before because without the *hits*, there are fewer candidates to receive ILP weight. Ideally, each miss could be assigned weight based on its average expected dynamic latency, but to effect this change would require a completely new implementation.

[3] The weight of the instruction is *not* a latency.

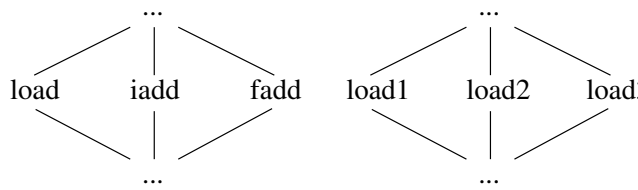


Fig. 3. Example: Balanced scheduling for multi issue processors.

4.3 Communicating Locality Classifications to the Scheduler

In this section, we describe how to translate our classification of *hits* and *misses* which are relative to the assembler code into the Multiflow's higher-level intermediate representation (IR). Using the internal representation before the scheduling pass, we add a unique tag to each load. After scheduling, when the compiler writes out the assembly code, it also writes the tags and load line number to a file. The locality analysis integrates these tags with the runtime information. When we recompile the program, the compiler uses the tags to map the locality information to the internal representation.

The locality analysis compares the Multiflow assembler and the executed assembler to find corresponding basic blocks. The assembler code output by the Multiflow is not complete, e.g., branch instructions and nops are missing. Some blocks have no locality data and some cannot be matched. These flaws result in no locality information for about 25% of all blocks. When we do not have or cannot map locality information, we classify loads as *misses* following the Balanced scheduler's conservative policy.

4.4 Limitations of Experiments

A systematic problem is that register assignment is performed after scheduling, which is true in many systems. We cannot use the locality data for spilled loads or any other loads that are inserted after scheduling because these loads do not exist in the scheduler's internal representation, and different spills are of course required for different schedules. Unfortunately these loads are a considerable fraction of all loads. The fraction of spilled loads for our benchmarks appear in the second and sixth columns of Table 1. `Apsi` spills 44.9% of all loads, and `turb3d` spills 47.7%. A scheduler that runs after register assignment would avoid this problem, but introduces the problem that register assignment reduces the available ILP.

The implementation of the Balanced scheduling algorithm we use is tuned for a single issue machine. If an instruction can be placed behind an other one the other's weight is increased, without considering whether a cycle can be hidden at all; i.e., the instruction could be issued in parallel with a second one placed behind that other instruction. Figure 3 shows two simple DAGs. In the left DAG, the floating point and the integer add both may issue in the delay slot of the load, and the scheduler thus increases the weight of the load by one for each add. On a single issue machine, this weighting correctly suggests that two cycles load latency can be hidden. On the Alpha 21164, all

three instructions may issue in parallel[4], i.e., placing the adds in the delay slot does not hide the latency of the load. Similarly in the DAG on the right, the Balanced scheduler will give `load1` a weight of 2. Here only one cycle of the latency can be hidden, because only one of the other loads can issue in parallel. The weight therefore does not correctly represent how many cycles of latency can be hidden, but instead how many instructions may issue behind it.

Another implementation problem is that the Balanced scheduler assumes a static load latency of 1 cycle, whereas the machines we use have a 2 or 3 cycle load delay. Since the scheduler covers the static latencies of non-loads first, if there is limited ILP the static latencies of loads may not be covered (as we mentioned in Section 4.1). When we classify some loads as *hits*, we exacerbate this problem because now neither pass assigns these loads any weight. We correct this problem by increasing the weights of all loads classified as *hits* to their static latency, 2 or 3 cycles, which the list scheduler will then hide if possible. This change however breaks the paradigm of Balanced scheduling, as weight is introduced that is not based on available ILP.

5 Experimental Results

We used the SPECfp95 benchmarks, one SPECint95 benchmark, and the Livermore loops in our experiments. The numbers for Livermore are for the whole benchmark with all kernels. We first compiled the programs with Balanced scheduling, overwriting the weights of loads with 1 to achieve a schedule where load latencies are not hidden. We executed this program several times and monitored it with DCPI to collect data for the locality analysis.

We then compiled each benchmark twice, once with Balanced scheduling, and a second time with Balanced scheduling and our locality data. We ran these programs five times on an idle machine, measured the runtimes with DCPI, and averaged the runtimes. The standard deviation of the runs is less than the differences we report. We used the same input in all runs and thus are reporting the upper bound on any expected improvements. The DCPI runtime numbers correspond to numbers generated with the operating system command `time`. We executed the whole experiment twice, once on an Alpha 21064 and once on a 21164. To show the sensitivity of scheduling to the quality of the locality data we use both the strict and the generous heuristic on the Alpha 21164. We expect our heuristic to perform better on the dual-issue 21064 than on the quad-issue 21164 because it needs less ILP to satisfy the issue width. The 21164 is of course more representative of modern processors.

Table 1 shows the number of loads we were able to analyze with the strict heuristic. The total number of loads includes only basic blocks with useful monitoring data, i.e., blocks that are executed several times. The first column gives the percentage of loads inserted during or after scheduling that we cannot use. The second column gives the percentage of loads for which the locality data cannot be evaluated because the instruction that uses it is not in the same block, or because the basic block could not be mapped on

[4] The Alpha 21164 can issue two integer and two floating point operations at once. Loads are integer operations with respect to this rule.

	21064				21164			
program	spill/all	nodata/all	hit/all	hit/anal	spill/all	nodata/all	hit/all	hit/anal
applu	12.9	57.8	13.7	46.7	24.6	46.5	7.9	27.3
apsi	29.2	27.9	16.6	38.6	44.9	17.5	8.6	22.9
fpppp	21.4	70.3	4.3	51.3	18.1	73.6	1.4	17.3
hydro2d	8.2	34.6	22.2	38.9	8.5	33.4	10.2	17.6
mgrid	13.5	42.0	22.4	50.3	15.1	41.5	12.5	28.7
su2cor	17.2	35.2	19.0	40.0	16.9	38.1	6.7	14.8
swim	4.6	25.7	21.1	30.3	5.5	5.5	1.4	1.5
tomcatv	1.8	69.0	6.3	21.7	6.2	33.3	0.7	1.1
turb3d	45.9	25.4	9.5	32.9	47.7	24.8	10.4	37.8
comprs.95	16.2	14.9	31.1	45.1	16.3	14.4	40.5	58.5
livermore	13.9	37.1	25.9	52.9	13.3	40.2	17.1	36.8

Table 1. Percentage of analyzed loads 21064 and 21164

the intermediate representation. Although we use the same binaries, `dcpicalc` produces different results on the different architectures and thus the sets of basic blocks may differ. The third column gives the percentage of loads the strict heuristic classifies as *hits* out of all the loads. It classifies all loads not appearing in columns 1-3 as *misses*. The last column gives the percentage of loads with useful locality data, i.e., those not appearing in columns 1 and 2.

On the 21164, our classification marks very few loads as *hits* for `swim` and `tomcatv`, and thus should have little effect. To address the problem that about half of all loads have no locality data, we need different tools, although additional sampling executions might help.

Table 2 gives relative performance numbers: Balanced scheduling with locality information divided by regular Balanced scheduling. The first two columns are for the strict heuristic which on average slightly improves the runtime of the benchmarks. The two columns give performance numbers for experiments on an Alpha 21064 and an Alpha 21164. The third column gives the performance of a program optimized with locality data produced by the generous heuristic, and executed on an Alpha 21164. On average, scheduling with the generous heuristic degrades performance slightly.

Many blocks have the same schedule in both versions and have only 1 or 2 instructions. 18% of the blocks with more than five instructions have no locality data available for rescheduling because the locality data could not be integrated into the compiler and therefore have identical schedules. 56% of the blocks where locality data is available have either no loads (other than spill loads), or all loads have been classified as *misses*. The remaining 26% of blocks have useful locality data available, and a different schedule. Therefore, the improvements stem from only a quarter of the program.

Although the average results are disappointing, improvements are possible. In two cases, we improve performance by 10% (`su2cor` on the 21064 and `fpppp` on the 21164), and these results are due to better scheduling. The significant degradations of two programs, (`compress95` on the 21064 and `turb3d` on the 21164), are due to flaws in the Balanced scheduler rather than inaccuracies the locality data introduces.

program	strict heuristic		gen. heu.
	21064	21164	
apsi	99.7	100.4	99.9
fpppp	98.1	90.6	101.1
hydro2d	100.4	99.4	101.9
mgrid	101.2	101.7	104.2
su2cor	90.2	99.6	100.4
swim	100.2	99.2	99.3
tomcatv	101.8	99.6	102.9
turb3d	96.8	106.3	105.1
compress95	107.6	98.8	99.5
livermore	100.3	98.7	96.9
average	99.6	99.4	101.1

Table 2. Performance of programs scheduled with locality data.

Further investigation on procedure and block level showed that mostly secondary effects of the tools spoiled the effect of our optimization; the optimization improved the performance of those blocks where these secondary effects played no role. Space constraints precludes explaining these details.

6 Conclusions

In this study, we have shown that it is possible to exploit the run-time information provided by hardware counters to tune applications. We have exploited the locality information provided by these counters to improve instruction scheduling. As it is still difficult to determine statically whether a load hits or misses frequently, hardware counters act as a natural complement to classic static optimizations. Because of the limitations of the scheduler tools we used, we could not exploit all the information provided by DCPI (miss ratio instead of latencies). We believe that our approach is promising, but that it needs new scheduling algorithms that take in to account variable latencies and issue width to be fully realized.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, June 1997.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 72–84, Montreal, Canada, June 1998.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[4] S. Carr. Combining optimization for cache and instruction-level parallelism. In *The 1996 International Conference on Parallel Architectures and Compilation Techniques*, Boston, MA, October 1996.

[5] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction level profiling on out-of-order processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[6] Chen Ding, Steve Carr, and Phil Sweany. Modulo scheduling with cache reuse information. In *Proceedings of EuroPar '97*, pages 1079–1083, August 1997.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[8] D. R. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, NM, June 1993.

[9] C. Liao, M. Martonosi, and D. W. Clark. Performance monitoring in a myrinet-connected shrimp cluster. In *1998 ACM Sigmetrics Symposium on Parallel and Distributed Tools*, August 1998.

[10] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 151–162, San Diego, CA, June 1995.

[11] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, pages 51–143, 1993.

[12] F. Jesus Sanchez and Antonio Gonzales. Cache sensitive modulo scheduling. In *The 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 261–271, November 1997.

[13] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.

Neighbourhood Preserving Load Balancing: A Self-Organizing Approach

Attila Gürsoy and Murat Atun

Computer Engineering Department,
Bilkent University,
Ankara Turkey
{agursoy, atun}@cs.bilkent.edu.tr

Abstract. We describe a static load balancing algorithm based on Kohonen Self-Organizing Maps (SOM) for a class of parallel computations where the communication pattern exhibits spatial locality and we present initial results. The topology preserving mapping achieved by SOM reduces the communication load across processors, however, it does not take load balancing into consideration. We introduce a load balancing mechanism into the SOM algorithm. We also present a preliminary multilevel implementation which resulted in significant execution time improvements. The results are promising to further improve SOM based load balancing for geometric graphs.

1 Introduction

A parallel program runs efficiently when its tasks are assigned to processors in such a way that load of every processor is more or less equal, and at the same time, amount of communication between processors is minimized. In this paper, we discuss a static load balancing heuristic based on Kohonen's self-organizing maps (SOM) [1] for a class of parallel computations where the communication pattern exhibits spatial locality. Many parallel scientific applications including molecular dynamics, fluid dynamics, and others which require solving numerical partial differential equations have this kind of communication pattern. In such applications, the physical problem domain is represented with a collection of nodes of a graph where the interacting nodes are connected with edges. In order to perform these computations on a parallel machine, the tasks (the nodes of the graph) need to be distributed to processor. Balancing the load of processors requires the computational load (total weight of the nodes) to be evenly distributed to processors and at the same time the communication overhead (which corresponds to edges connecting nodes assigned to different processors) to be minimized. In this paper, we are interested in static load balancing problem, that is, the computational load of the tasks can be estimated a priori and the computation graph does not change rapidly during the execution. However, the approach can be extended to dynamic load balancing easily.

The partitioning and mapping of tasks of a parallel program to minimize the execution time is a hard problem. Various approaches and heuristics have

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 234–241, 2000.
© Springer-Verlag Berlin Heidelberg 2000

been developed to solve this problem. Most approaches are for arbitrary computational graphs such as the heuristic of Kernighan-Lin [2] which is a graph bipartitioning approach with minimal cut costs and the ones based on physical or stochastic optimization such as simulated annealing and neural networks [3]. The computational graphs that we are interested, on the other hand, have spatially localized communication patterns. For example, the computational graph from a molecular dynamics simulation [4] is such that the nodes correspond to particles, and the interactions of a particle is limited to physically close particles. In these cases, it is sometimes desirable to partition the computation graph spatially not only for load balancing purposes but for also other algorithmic or programming purposes. This spatial relation can also be exploited to achieve an efficient partitioning of the graph. The communication overhead can be reduced if the physically nearby and heavily communicating tasks are mapped to the same processor or to the same group of processors. The popular methods are based on decomposing the computation space recursively such as the recursive coordinate bisection method. However this simple scheme fails under certain cases. More advanced schemes include the nearest neighbour mapping heuristic [5] and partitioning by space filling curves [6] which try to exploit the locality of communication of the computation graph. In this work, we will present an algorithm based on Kohonen's SOM to partition such graphs. The idea of SOM algorithm is originated from the organizational structure of human brain and the learning mechanisms of biological neurons. After a training phase, the neurons become organized in such a way that they reflect the topological properties of the input set used in training. This important property of SOM — topology preserving mapping — makes it an ideal candidate for partitioning geometric graphs. We propose an algorithm based on SOM to achieve load balancing. Applying self-organization to partitioning and mapping of parallel programs has been discussed by a few researchers [7], [8], however, our modeling and incorporation of load balancing into SOM is quite different from those work and the experiments showed that our algorithm achieves load balancing more effectively.

The rest of the paper is organized as follows: In Section 2, we give a brief description of the Kohonen maps. Then, we describe a load balancing algorithm based on SOM and present its performance. In Section 4, we present a preliminary multilevel approach to further improve the execution time of the algorithm and discuss future work in the last section.

2 Self Organizing Maps (SOM)

The Kohonen's self-organizing map is a special type of competitive learning neural network. The basic architecture of Kohonen's map is n neurons that are generally connected in a d-dimensional space, for example a grid, where each neuron is connected with its neighbours. The map has two layers: an input layer and an output layer which consists of neural units. Each neuron in the output layer is connected to every input unit. A weight vector w_i is associated with

each neuron i. An input vector, v, which is chosen according to a probability distribution is forwarded to the neuron layer during the competitive phase of the SOM. Every neuron calculates the difference of its weight vector with the input vector v (using a prespecified distance measure, for example, Euclidean distance if the weight and input vectors represent points in space). The neuron with the smallest difference wins the competition and is selected to be the excitation center c:

$$||w_c - v|| = \min_{k=1}^{n} ||w_k - v||$$

After the excitation center is determined, the weight vectors of the winner neuron and its topological neighbours are updated so as to align them towards the input vector. This step corresponds to the cooperative learning phase of the SOM. The learning function is generally formulated as:

$$w_i \leftarrow w_i + \epsilon * e^{\frac{-d(c,i)}{2\theta^2}} * ||w_i - v||$$

The Kohonen algorithm has two important execution parameters. These are ϵ and θ. ϵ is the learning rate. Generally, it is a small value varying between 1 and 0. It may be any decreasing function with increasing time step or a constant value. θ is the other variable which highly controls the convergence of the Kohonen algorithm. It determines the set of neurons to be updated at each step. Only the neurons within the neighbourhood defined by θ is updated by an amount depending on the distance $d(c, i)$. θ is generally an exponential decreasing function with respect to increasing time step.

3 Load Balancing with SOM

In this section, we will present a SOM based load balance algorithm. For the sake of simplicity, the discussion is limited to two dimensional graphs. We assume that the nodes of the graph might have different computational loads but the communication load per edge is uniform. In addition, we assume that the cost of communicating between any pair of processors is similar (this is a realistic assumption since most recent parallel machines has wormhole routing). However, it is easy to extend to model to cover more complicated cases. As far as the partitioning of a geometric graph is considered, the most important feature of the SOM is that it achieves a topology-representing mapping. Let the unit square $S = [0, 1]^2$ be the input space of the self-organizing map. We divide S into p regions called processor regions where $p = p_x \times p_y$ is the number of processors. Every processor P_{ij} has a region, S_{ij}, of coordinates which is a subset of S bounded by $i \times \text{width}_x, j \times \text{width}_y$ and $(i + 1) \times \text{width}_x, (j + 1) \times \text{width}_y$ where $\text{width}_x = 1/p_x$ and $\text{width}_y = 1/p_y$. Let each node (or task) of the computation graph (that we want to partition and map to processors) correspond to a neuron in the self-organizing map. That is, the computation graph corresponds to the neural layer. A neuron is connected to other neurons if they are connected in the computation graph. We define the weight vectors of the neurons to be the

Algorithm 1 Load Balancing using SOM

```
for all neurons i do
    initialize weight vectors w_i = (x, y) ∈ S randomly
end for
for all processors i do
    calculate load of each processor
end for
set initial and final values of diameter θ_i and θ_f
set initial and final values of learning constant ε_i and ε_f
for t = 0 to t_max do
    let S_p be the region of the least loaded processor p
    select a random input vector v = (x, y) ∈ S_p
    determine the excitation center c such that for all neurons n
    ||w_c − v|| = min ||w_n − v||
    for d = 0 to θ do
        for all neurons k with distance d from center c do
            update weight vectors w_k ← w_k + ε e^(−d/(2θ²)) ||w_k − v||
        end for
    end for
    update diameter θ ← θ_i (θ_f/θ_i)^(t/t_max)
    update learning constant ε ← ε_i (ε_f/ε_i)^(t/t_max)
    update load of each processor
end for
```

positions on the unit square S. That is, each weight vector, $w = (x, y)$, is a point in S. Now, we define also mapping of a task to a processor as follows: A task t is mapped to a processor P_{ij} if $w_t \in S_{ij}$.

The load balancing algorithm, Algorithm1, starts with initializing various parameters. First, all tasks are distributed to processors randomly (that is, weight vectors are initialized to random points in S). During the learning phase of the algorithm, we repetitively chose an input vector from S and present it to the neural layer. If we choose the input vector with uniform probability over S, then, the neural units will try to fill the area S accordingly. If the computation of each task (node) and communication volume (edges) are uniform (equal), then the load balance of this mapping will be near-optimal. However, most computational graphs might have non-uniform computational loads at each node. In order to enforce a load balanced mapping of tasks, we have to select input vectors differently. This can be achieved by selecting inputs from the regions closer to the least loaded processor. This strategy will probably force SOM to shift the tasks towards to least loaded processor and the topology preserving feature of SOM will try to keep communication minimum. A detailed study of how to choose input vector and various alternatives is presented in [9]. It has been experimentally found out that choosing the input vector always in the region of the least loaded processor leads to better results. As we mentioned before, ϵ is the learning rate which is generally an exponential function decreasing with increasing time step. At the initial steps of the algorithm, ϵ is closer to 1, which means learning

rate is high. Towards to the end, as ϵ becomes closer to 0, the adjustments do minor changes on weight vectors and so the mapping becomes more stable. In our algorithm we used 0.8 and 0.2 for initial and final values of ϵ. To determine the set of neurons to be updated, we defined θ to be the length of the shortest path (number of edges) to reach from the excitation center. Initially, it has a value of $\theta_i = \sqrt{n}$ and it exponentially decreases to 1. These values are the most common choices used in SOMs. The lines 13-17 in Algorithm1 correspond to this update process. Figure 1 illustrates a partitioning of a graph with 4253 nodes into eight processors using the proposed algorithm.

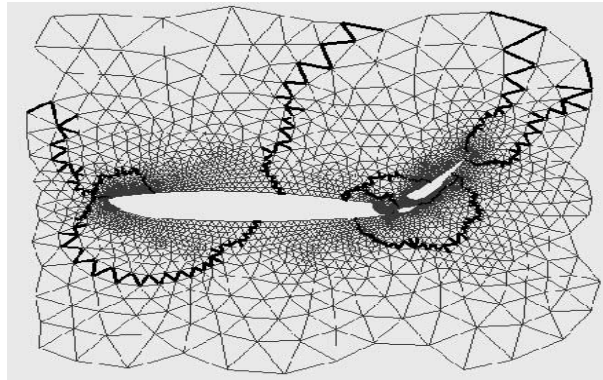


Fig. 1. Partitioning a FEM graph (airfoil): 4253 nodes to 8 processor

3.1 Results

We have tested our algorithm on some known FEM/Grid graphs available from AG-Monien (Parallel Computing Group) Web pages [10]. We compared the performance of our algorithm with the results of algorithm given in Heiss-Dormanns [7]. They reported that their SOM based load balancing algorithm was comparable with other approaches. Particularly, the execution time of their algorithm was on the average larger than mean field annealing based solutions but less than simulated annealing ones for random graphs.

We conducted runs on a set of FEM/Grid graphs and gathered execution times for our algorithms and Heiss-Dormanns' algorithm on a Sun Ultra2 workstation with 167MHZ processor. Table 1 shows the load balance achieved and total execution times. Our approach performed better on all cases. However, as in other stochastic optimization problems, the selection of various parameters such as learning rate plays and important role in the performance. For the runs for the algorithm by Heiss-Dormanns, we used their suggestions for setting various parameters in the algorithm given in their reports.

Table 1. Load balance and execution time results for FEM/Grid graphs

		Communication Cost(x1000)		Load Imbalance (%)		Execution Time (secs)	
Graph	Processor Mesh	Heiss-Dorm. Alg.	Our Alg.	Heiss-Dorm. Alg.	Our Alg.	Heiss-Dorm. Alg.	Our Alg.
3elt	4X4	2.16	1.11	22.71	0.45	604.19	135.69
3elt	4X8	1.25	1.66	15.03	1.47	372.69	133.08
Airfoil	4X4	1.76	1.04	24.15	0.57	498.66	118.67
Airfoil	4X8	0.90	1.56	10.04	0.82	300.22	115.98
Bcspwr10	4X4	0.48	0.76	21.96	0.33	555.96	162.65
Bcspwr10	4X8	0.74	1.09	51.14	2.44	862.85	159.21
Crack	4X4	2.23	1.51	5.47	0.21	1986.90	290.65
Crack	4X8	3.67	2.37	26.88	0.42	810.02	281.52
Jagmesh	4X4	0.50	0.39	12.25	0.85	16.87	18.26
Jagmesh	4X8	0.92	0.62	32.19	4.84	29.27	18.40
NASA4704	4X4	136.02	10.27	60.77	0.91	843.27	214.34
NASA4704	4X8	255.31	14.90	88.21	3.63	1182.2	211.09

4 Improvement with Multilevel Approach

In order to improve the execution time performance of the load balancing algorithm, we modified it to do the partitioning in a multilevel way. Since physically nearby nodes get assigned to the same processor (most likely), it will be beneficial if we could cluster a group of nodes into a super node and run the load balancing on this coarser graph, then unfold the super nodes and refine the partitioning. This is very similar to multilevel graph partitioning algorithms which have been used very successfully to improve the execution time [11]. In multilevel graph partitioning, the initial graph is coarsened, to get smaller graphs where a node in the coarsened graph represents a set of nodes in the original graph. When the graph is coarsened to a reasonable size, an expensive but powerful partitioner performs an initial partitioning of the coarsened graph. Then, the graph is uncoarsened, that is unfolding the collapsed nodes, and mapping of unfolded nodes is handled (refinement phase). In our implementation, we have used the heavy-edge-matching (HEM) scheme for the coarsening phase as described in [11]. HEM scheme selects nodes in random. If a node has not matched yet, then the node is matched with a maximum weight neighbour node. This algorithm is applied iteratively, each time obtaining a coarser graph. For initial partitioning of the coarsest graph and refinement phases, we have used our SOM algorithm without any change. The performance results of a preliminary multilevel implementation of our algorithm for the FEM/Grid graphs is presented in Table 2. The results show that multilevel implementation reduces the execution time significantly.

Table 2. Execution results of SOM and MSOM: n-initials is the number of nodes in the initial graph, n-final is the number of nodes in the coarsest graph, and Levels is the number of coarsening levels

Graph	Processors	n-initial	Levels	n-final	Load Imbalance		Execution Time	
					SOM	MSOM	SOM	MSOM
Whitaker	4x4	9800	9	89	0.08	0.73	216.18	42.29
Whitaker	4x8	9800	9	89	0.57	1.55	212.16	60.57
Jagmesh	4x4	936	5	65	0.85	1.42	18.26	3.86
Jagmesh	4x8	936	5	65	4.84	5.98	18.40	5.70
3elt	4x4	4720	8	71	0.45	1.69	135.69	31.99
3elt	4x8	4720	8	71	1.47	1.69	133.08	33.95
Airfol	4x4	4253	8	63	0.57	1.07	118.67	28.58
Airfol	4x8	4253	8	63	0.82	1.82	115.98	35.16
NASA704	4x4	4704	7	97	0.91	4.98	214.34	61.31
NASA704	4x8	4704	7	97	3.63	8.16	211.09	101.28
Big	4x4	4704	10	96	0.10	2.25	521.28	78.50
Big	4x8	4704	10	96	4.37	2.66	492.30	142.69

5 Related Work

Heiss-Dormanns used computation graph as input space and processors as output space (the opposite of our algorithm). A load balancing correction, activated once per a predetermined number of steps, changes the receptive field of processor nodes according to their load. Changing the magnitude of a receptive field corresponds to transferring the loads between these receptive fields. The results show that our approach handles load balancing better and has better execution time performance.

In another SOM based work, Meyer [8] identified the computation graph with the output space and processors with input space (called inverse mapping in their paper). Load balancing was handled by defining a new distance metric to be used in learning function. According to this new metric the shortest distance between any two vertices in the output space is formed by the vertices of least loaded ones of all other paths. It is reported that SOM based algorithm performs better than simulated annealing approaches.

6 Conclusion

We describe a static load balancing algorithm based on Self-Organizing Maps (SOM) for a class of parallel computations where the communication pattern exhibits spatial locality. It is sometimes desirable to partition the computation graph spatially not only for load balancing purposes but for also other algorithmic or programming purposes. This spatial relation can also be exploited to achieve an efficient partitioning of the graph. The communication overhead can

be reduced if the physically nearby and heavily communicating tasks are mapped to the same processor or to the same group of processors. The important property of SOM — topology preserving mapping — makes it an interesting approach for such partitioning. We represented tasks (nodes of the computation graph) as neurons and processors as the input space. We enforced load balancing by choosing input vectors from the region of least loaded processor. Also, a preliminary multilevel implementation is discussed which has improved the execution time significantly. The results are very promising (it produced better results than the other self-organized approaches). As future work, we plan to work on improving both the performance of the current implementation and also develop new multilevel coarsening and refinement approaches for SOM based partitioning.

Acknowledgement
We thank H.Heiss and M. Dormanns for providing us the source code of their implementation.

References

1. Kohonen, T.: The Self-Organizing Map *Proc. of the IEEE, Vol.78, No.9, September*, 1990, pp.1464-1480
2. Kernighan, B.W., Lin., S.: An Efficient Heuristic for Partitioning Graphs, Bell Syst. J., 49, 1970, pp. 291-307
3. Bultan T., Aykanat C.: A New Mapping Heuristic Based on Mean Field Annealing, J. Parallel Distrib. Comput., 1995, vol 16, pp. 452-469
4. Nelson, M., et al.: NAMD: A Parallel Object-Oriented Molecular Dynamics Program, Intn. Journal of Supercomputing Applications and High Performance Computing, Volume 10, No.4, 1996., pp.251-268
5. Sadayappan, P., Ercal., F.: Nearest-Neighbour Mapping of Finite Element Graphs onto Processor Meshes, IEEE Trans. on Computers, Vol. C-36, No 12, 1987, pp. 1408-1424
6. Pilkington, J.R, Baden, S.B.: Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves, IEEE Trans. on Parallel and Distributed Sys. 1997, Volume 7, pp. 288-300
7. Heiss, H., Dormanns, M.: Task Assignment by Self-Organizing Maps *Interner Bericht Nr.17/93, Mai 1993* Universität Karlsruhe, Fakultät für Informatik
8. Quittek, J.W., Optimizing Parallel Program Execution by Self-Organizing Maps, Journal of Artificial Neural Networks, Vol.2, No.4, 1995, pp.365-380
9. Atun, M.: A New Load Balancing Heuristic Using Self-Organizing Maps, M.Sc Thesis, Computer Eng. Dept., Bilkent University, Ankara, Turkey, 1999
10. University of Paderborn, AG-Monien Home Page (Parallel Computing Group), http://www.uni-paderborn.de/fachbereich/AG/monien.
11. Karypis., G, Kumar. V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, TR 95-035, Department of Computer Science, University of Minesota, 1995.

The Impact of Migration on Parallel Job Scheduling for Distributed Systems

Yanyong Zhang[1], Hubertus Franke[2], Jose E. Moreira[2], and Anand Sivasubramaniam[1]

[1] Department of Computer Science & Engineering
The Pennsylvania State University
University Park PA 16802
{yyzhang, anand}@cse.psu.edu
[2] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh, jmoreira}@us.ibm.com

Abstract. This paper evaluates the impact of task migration on gang-scheduling of parallel jobs for distributed systems. With migration, it is possible to move tasks of a job from their originally assigned set of nodes to another set of nodes, during execution of the job. This additional flexibility creates more opportunities for filling holes in the scheduling matrix. We conduct a simulation-based study of the effect of migration on average job slowdown and wait times for a large distributed system under a variety of loads. We find that migration can significantly improve these performance metrics over an important range of operating points. We also analyze the effect of the cost of migrating tasks on overall system performance.

1 Introduction

Scheduling strategies can have a significant impact on the performance characteristics of large parallel systems [3, 5, 6, 7, 12, 13, 17]. When jobs are submitted for execution in a parallel system they are typically first organized in a job queue. From there, they are selected for execution by the scheduler. Various priority ordering policies (FCFS, best fit, worst fit, shortest job first) have been used for the job queue. Early scheduling strategies for distributed systems just used a space-sharing approach, wherein jobs can run side by side on different nodes of the machine at the same time, but each node is exclusively assigned to a job. When there are not enough nodes, the jobs in the queue simply wait. Space sharing in isolation can result in poor utilization, as nodes remain empty despite a queue of waiting jobs. Furthermore, the wait and response times for jobs with an exclusively space-sharing strategy are relatively high [8].

Among the several approaches used to alleviate these problems with space sharing scheduling, two have been most commonly studied. The first is a technique called backfilling, which attempts to assign unutilized nodes to jobs that

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 242–251, 2000.
© Springer-Verlag Berlin Heidelberg 2000

are behind in the priority queue (of waiting jobs), rather than keep them idle. A lower priority job can be scheduled before a higher priority job as long as it does not delay the start time of that job. This requirement of not delaying higher priority jobs imposes the need for an estimate of job execution times. It has already been shown [4, 13, 18] that a FCFS queueing policy combined with backfilling results in efficient and fair space sharing scheduling. Furthermore, [4, 14, 18] have shown that overestimating the job execution time does not significantly change the final result.

The second approach is to add a time-sharing dimension to space sharing using a technique called gang-scheduling or coscheduling [9]. This technique virtualizes the physical machine by slicing the time axis into multiple space-shared virtual machines [3, 15], limited by the maximum multiprogramming level (MPL) allowed in the system. The schedule is represented as a cyclical Ousterhout matrix that defines the tasks executing on each processor and each time-slice. Tasks of a parallel job are coscheduled to run in the same time-slices (same virtual machines). A cycle through all the rows of the Ousterhout matrix defines a *scheduling cycle*.

Gang-scheduling and backfilling are two optimization techniques that operate on orthogonal axes, space for backfilling and time for gang-scheduling. The two can be combined by treating each of the virtual machines created by gang scheduling as a target for backfilling. We have demonstrated the efficacy of this approach in [18].

The approaches we described so far adopt a static model for space assignment. That is, once a job is assigned to nodes of a parallel system it cannot be moved. We want to examine whether a more dynamic model can be beneficial. In particular, we look into the issue of *migration*, which allows jobs to be moved from one set of nodes to another, possibly overlapping, set [16]. Implementing migration requires additional infrastructure in many parallel systems, with an associated cost. Migration requires significant library and operating system support and consumes resources (memory, disk, network) at the time of migration [2].

This paper addresses the following issues which help us understand the impact of migration. First, we determine if there is an improvement in the system performance metrics from applying migration, and we quantify this improvement. We also quantify the impact of the cost of migration (*i.e.*, how much time it takes to move tasks from one set of nodes to another) on system performance. Finally, we compare improvements in system performance that come from better scheduling techniques, backfilling in this case, and improvements that come from better execution infrastructure, as represented by migration. We also show the benefits from combining both enhancements.

The rest of this paper is organized as follows. Section 2 describes the migration algorithm we use. Section 3 presents our evaluation methodology for determining the quantitative impact of migration. In Section 4, we show the results from our evaluation and discuss the implications. Finally, Section 5 presents our conclusions.

2 The Migration Algorithm

Our scheduling strategy is designed for a distributed system, in which each node runs its own operating system image. Therefore, once tasks are started in a node, it is preferable to keep them there. Our basic (nonmigration) gang-scheduling algorithm, both with and without backfilling, works as follows. At every scheduling event (*i.e.*, job arrival or departure) a new scheduling matrix is derived:

- We schedule the already executing jobs such that each job appears in only one row (*i.e.*, into a single virtual machine). Jobs are scheduled on the same set of nodes they were running before. That is, no migration is attempted.
- We *compact* the matrix as much as possible, by scheduling multiple jobs in the same row. Without migration, only nonconflicting jobs can share the same row. Care must be taken in this phase to ensure forward progress. Each job must run at least once during a scheduling cycle.
- We then attempt to schedule as many jobs from the waiting queue as possible, using a FCFS traversal of that queue. If backfilling is enabled, we can look past the first job that cannot be scheduled.
- Finally, we perform an *expansion phase*, in which we attempt to fill empty holes left in the matrix by replicating job execution on a different row (virtual machine). Without migration, this can only be done if the entire set of nodes used by the job is free in that row.

The process of migration embodies moving a job to any row in which there are enough free processors. There are basically two options each time we attempt to migrate a job A from a source row r to a target row p (in either case, row p must have enough nodes free):

- *Option 1*: We migrate the jobs which occupy the nodes of job A at row p, and then we simply replicate job A, in its same set of nodes, in row p.
- *Option 2*: We migrate job A to the set of nodes in row p that are free. The other jobs at row p remain undisturbed.

We can quantify the cost of each of these two options based on the following model. For the distributed system we target, namely the IBM RS/6000 SP, migration can be accomplished with a checkpoint/restart operation. (Although it is possible to take a more efficient approach of directly migrating processes across nodes [1, 10, 11], we choose not to take this route.) Let $S(A)$ be the set of jobs in target row p that overlap with the nodes of job A in source row r. Let C be the total cost of migrating one job, including the checkpoint and restart operations. We consider the case in which (i) checkpoint and restart have the same cost $C/2$, (ii) the cost C is independent of the job size, and (iii) checkpoint and restart are dependent operations (*i.e.*, you have to finish checkpoint before you can restart). During the migration process, nodes participating in the migration cannot make progress in executing a job. We call the total amount of resources (processor $\times$ time) wasted during this process *capacity loss*. The capacity loss for option 1 is

$$(\frac{C}{2} \times |A| + C \times \sum_{J \in S(A)} |J|), \tag{1}$$

where $|A|$ and $|J|$ denote the number of tasks in jobs A and J, respectively. The loss of capacity for option 2 is estimated by

$$(C \times |A| + \frac{C}{2} \times \sum_{J \in S(A)} |J|). \tag{2}$$

The first use of migration is during the compact phase, in which we consider migrating a job when moving it to a different row. The goal is to maximize the number of empty slots in some rows, thus facilitating the scheduling of large jobs. The order of traversal of jobs during compact is from least populated row to most populated row, wherein each row the traversal continues from smallest job (least number of processors) to largest job. During the compact phase, both migration options discussed above are considered, and we choose the one with smaller cost.

We also apply migration during the expansion phase. If we cannot replicate a job in a different row because its set of processors are busy with another job, we attempt to move the blocking job to a different set of processors. A job can appear in multiple rows of the matrix, but it must occupy the same set of processors in all the rows. This rule prevents the ping-pong of jobs. For the expansion phase, jobs are traversed in first-come first-serve order. During expansion phase, only migration option 1 is considered.

As discussed, migration in the IBM RS/6000 SP requires a checkpoint/restart operation. Although all tasks can perform a checkpoint in parallel, resulting in a C that is independent of job size, there is a limit to the capacity and bandwidth that the file system can accept. Therefore we introduce a parameter Q that controls the maximum number of tasks that can be migrated in any time-slice.

3 Methodology

Before we present the results from our studies we first need to describe our methodology. We conduct a simulation based study of our scheduling algorithms using synthetic workloads. The synthetic workloads are generated from stochastic models that fit actual workloads at the ASCI Blue-Pacific system in Lawrence Livermore National Laboratory (a 320-node RS/6000 SP). We first obtain one set of parameters that characterizes a specific workload. We then vary this set of parameters to generate a set of nine different workloads, which impose an increasing load on the system. This approach, described in more detail in [5, 18], allows us to do a sensitivity analysis of the impact of the scheduling algorithms over a wide range of operating points.

Using event driven simulation of the various scheduling strategies, we monitor the following set of parameters: (1) t_i^a: arrival time for job i, (2) t_i^s: start time for job i, (3) t_i^e: execution time for job i (on a dedicated setting), (4) t_i^f: finish

time for job i, (5) n_i: number of nodes used by job i. From these we compute: (6) $t_i^r = t_i^f - t_i^a$: response time for job i, (7) $t_i^w = t_i^s - t_i^a$: wait time for job i, and (8) $s_i = \frac{\max(t_i^r, T)}{\max(t_i^e, T)}$: the slowdown for job i, where T is the time-slice for gang-scheduling. To reduce the statistical impact of very short jobs, it is common practice [4] to adopt a minimum execution time. We adopt a minimum of one time slice. That is the reason for the $\max(\cdot, T)$ terms in the definition of slowdown.

To report quality of service figures from a user's perspective we use the average job slowdown and average job wait time. Job slowdown measures how much slower than a dedicated machine the system appears to the users, which is relevant to both interactive and batch jobs. Job wait time measures how long a job takes to start execution and therefore it is an important measure for interactive jobs.

We measure quality of service from the system's perspective with utilization. Utilization is the fraction of total system resources that are actually used for the execution of a workload. It does not include the overhead from migration. Let the system have N nodes and execute m jobs, where job m is the last job to finish execution. Also, let the first job arrive at time $t = 0$. Utilization is then defined as

$$\rho = \frac{\sum_{i=1}^{m} n_i t_i^e}{N \times t_m^f \times \mathrm{MPL}}. \tag{3}$$

For the simulations, we adopt a time slice of $T = 200$ seconds, multiprogramming levels of 2, 3, and 5, and consider four different values of the migration cost C: 0, 10, 20, and 30 seconds. The cost of 0 is useful as a limiting case, and represents what can be accomplished in more tightly coupled single address space systems, for which migration is a very cheap operation. Costs of 10, 20, and 30 seconds represent 5, 10, and 15% of a time slice, respectively.

To determine what are feasible values of the migration cost, we consider the situation that we are likely to encounter in the next generation of large machines, such as the IBM ASCI White. We expect to have nodes with 8 GB of main memory. If the entire node is used to execute two tasks (MPL of 2) that averages to 4 GB/task. Accomplishing a migration cost of 30 seconds requires transferring 4 GB in 15 seconds, resulting in a per node bandwidth of 250 MB/s. This is half the bandwidth of the high-speed switch link in those machines. Another consideration is the amount of storage necessary. To migrate 64 tasks, for example, requires saving 256 GB of task image. Such amount of fast storage is feasible in a parallel file system for machines like ASCI White.

4 Experimental Results

Table 1 summarizes some of the results from migration applied to gang-scheduling and backfilling gang-scheduling. For each of the nine workloads (numbered from 0 to 8) we present achieved utilization (ρ) and average job slowdown (s) for four different scheduling policies: (i) backfilling gang-scheduling without migration (BGS), (ii) backfilling gang-scheduling with migration (BGS+M), (iii)

gang-scheduling without migration (GS), and (iv) gang-scheduling with migration (GS+M). We also show the percentage improvement in job slowdown from applying migration to gang-scheduling and backfilling gang-scheduling. Those results are from the best case for each policy: 0 cost and unrestricted number of migrated tasks, with an MPL of 5.

We can see an improvement from the use of migration throughout the range of workloads, for both gang-scheduling and backfilling gang-scheduling. We also note that the improvement is larger for mid-to-high utilizations between 70 and 90%. Improvements for low utilization are less because the system is not fully stressed, and the matrix is relatively empty. Therefore, there are not enough jobs to fill all the time-slices, and expanding without migration is easy. At very high loads, the matrix is already very full and migration accomplishes less than at mid-range utilizations. Improvements for backfilling gang-scheduling are not as impressive as for gang-scheduling. Backfilling gang-scheduling already does a better job of filling holes in the matrix, and therefore the potential benefit from migration is less. With backfilling gang-scheduling the best improvement is 45% at a utilization of 94%, whereas with gang-scheduling we observe benefits as high as 90%, at utilization of 88%.

We note that the maximum utilization with gang-scheduling increases from 85% without migration to 94% with migration. Maximum utilization for backfilling gang-scheduling increases from 95% to 97% with migration. Migration is a mechanism that significantly improves the performance of gang-scheduling without the need for job execution time estimates. However, it is not as effective as backfilling in improving plain gang-scheduling. The combination of backfilling and migration results in the best overall gang-scheduling system.

work load	backfilling gang scheduling					gang scheduling				
	BGS		BGS+M		% s	GS		GS+M		% s
	ρ	s	ρ	s	better	ρ	s	ρ	s	better
0	0.55	2.5	0.55	2.4	5.3%	0.55	2.8	0.55	2.5	11.7%
1	0.61	2.8	0.61	2.6	9.3%	0.61	4.4	0.61	2.9	34.5%
2	0.66	3.4	0.66	2.9	15.2%	0.66	6.8	0.66	4.3	37.1%
3	0.72	4.4	0.72	3.4	23.2%	0.72	16.3	0.72	8.0	50.9%
4	0.77	5.7	0.77	4.1	27.7%	0.77	44.1	0.77	12.6	71.3%
5	0.83	9.0	0.83	5.4	40.3%	0.83	172.6	0.83	25.7	85.1%
6	0.88	13.7	0.88	7.6	44.5%	0.84	650.8	0.88	66.7	89.7%
7	0.94	24.5	0.94	13.5	44.7%	0.84	1169.5	0.94	257.9	77.9%
8	0.95	48.7	0.97	42.7	12.3%	0.85	1693.3	0.94	718.6	57.6%

Table 1. Percentage improvements from migration.

Figure 1 shows average job slowdown and average job wait time as a function of the parameter Q, the maximum number of task that can be migrated in any time slice. We consider two representative workloads, 2 and 5, since they define the bounds of the operating range of interest. Beyond workload 5, the system

reaches unacceptable slowdowns for gang-scheduling, and below workload 2 there is little benefit from migration. We note that migration can significantly improve the performance of gang-scheduling even with as little as 64 tasks migrated. (Note that the case without migration is represented by the parameter $Q = 0$ for number of migrated tasks.) We also observe a monotonic improvement in slowdown and wait time with the number of migrated tasks, for both gang-scheduling and backfilling gang-scheduling. Even with migration costs as high as 30 seconds, or 15% of the time slice, we still observe benefit from migration. Most of the benefit of migration is accomplished at $Q = 64$ migrated tasks, and we choose that value for further comparisons. Finally, we note that the behaviors of wait time and slowdown follow approximately the same trends. Thus, for the next analysis we focus on slowdown.

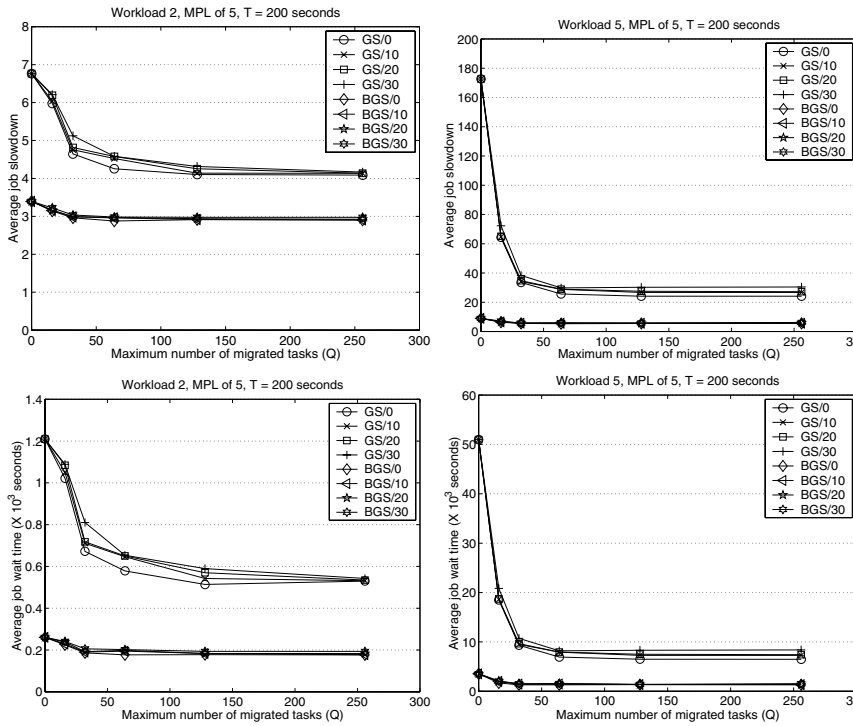


Fig. 1. Slowdown and wait time as a function of number of migrated tasks. Each line is for a combination of scheduling policy and migration cost.

Figure 2 shows average job slowdown as a function of utilization for gang-scheduling and backfilling gang-scheduling with different multiprogramming levels. The upper left plot is for the case with no migration ($Q = 0$), while the other plots are for a maximum of 64 migrated tasks ($Q = 64$), and three different mi-

gration costs, $C = 0$, $C = 20$, and $C = 30$ seconds, corresponding to 0, 10, and 15% of time slice, respectively. We observe that, in agreement with Figure 1, the benefits from migration are essentially invariant with the cost in the range we considered (from 0 to 15% of the time slice).

From a user perspective, it is important to determine the maximum utilization that still leads to an acceptable average job slowdown (we adopt $s \leq 20$ as an acceptable value). Migration can improve the maximum utilization of gang-scheduling by approximately 8%. (From 61% to 68% for MPL 2, from 67% to 74% for MPL 3, and from 73% to 81% for MPL 5.) For backfilling gang-scheduling, migration improves the maximum acceptable utilization from from 91% to 95%, independent of the multiprogramming level.

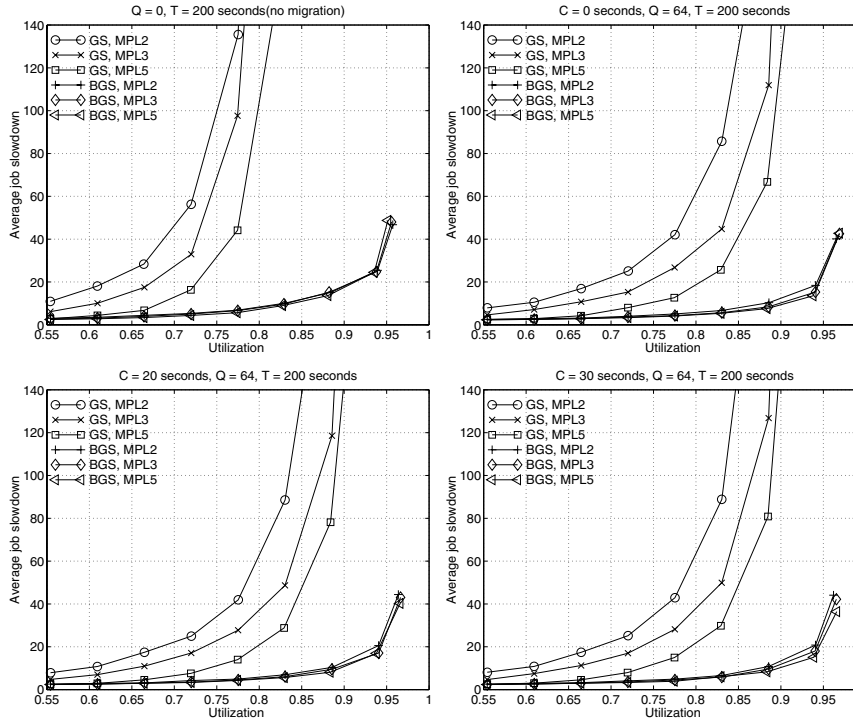


Fig. 2. Slowdown as function of utilization. Each line is for a combination of scheduling policy and multiprogramming level.

5 Conclusions

In this paper we have evaluated the impact of migration as an additional feature in job scheduling mechanisms for distributed systems. Typical job scheduling for

distributed systems uses a static assignment of tasks to nodes. With migration we have the additional ability to move some or all tasks of a job to different nodes during execution of the job. This flexibility facilitates filling holes in the schedule that would otherwise remain empty. The mechanism for migration we consider is checkpoint/restart, in which tasks have to be first vacated from one set of nodes and then reinstantiated in the target set.

Our results show that there is a definite benefit from migration, for both gang-scheduling and backfilling gang-scheduling. Migration can lead to higher acceptable utilizations and to smaller slowdowns and wait times for a fixed utilization. The benefit is essentially invariant with the cost of migration for the range considered (0 to 15% of a time-slice). Gang-scheduling benefits more than backfilling gang-scheduling, as the latter already does a more efficient job of filling holes in the schedule. Although we do not observe much improvement from a system perspective with backfilling scheduling (the maximum utilization does not change much), the user parameters for slowdown and wait time with a given utilization can be up to 45% better. For both gang-scheduling and backfilling gang-scheduling, the benefit is larger in the mid-to-high range of utilization, as there is not much opportunity for improvements at either the low end (not enough jobs) or very high end (not enough holes). Migration can lead to a better scheduling without the need for job execution time estimates, but by itself it is not as useful as backfilling. Migration shows the best results when combined with backfilling.

References

[1] J. Casas, D. L. Clark, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. **MPVM: A Migration Transparent Version of PVM**. *Usenix Computing Systems*, 8(2):171–216, 1995.

[2] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. **A worldwide flock of Condors: Load sharing among workstation clusters**. *Future Generation Computer Systems*, 12(1):53–65, May 1996.

[3] D. G. Feitelson and M. A. Jette. **Improved Utilization and Responsiveness with Gang Scheduling**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, April 1997.

[4] D. G. Feitelson and A. M. Weil. **Utilization and predictability in scheduling the IBM SP2 with backfilling**. In *12th International Parallel Processing Symposium*, pages 542–546, April 1998.

[5] H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. **An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific**. In *Proceedings of SC99, Portland, OR*, November 1999. IBM Research Report RC21559.

[6] B. Gorda and R. Wolski. **Time Sharing Massively Parallel Machines**. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.

[7] H. D. Karatza. **A Simulation-Based Performance Analysis of Gang Scheduling in a Distributed System**. In *Proceedings 32nd Annual Simulation Symposium*, pages 26–33, San Diego, CA, April 11-15 1999.

[8] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette. **An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments**. In *Proceedings of SC98, Orlando, FL*, November 1998.

[9] J. K. Ousterhout. **Scheduling Techniques for Concurrent Systems**. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[10] S. Petri and H. Langendörfer. **Load Balancing and Fault Tolerance in Workstation Clusters – Migrating Groups of Communicating Processes**. *Operating Systems Review*, 29(4):25–36, October 1995.

[11] J. Pruyne and M. Livny. **Managing Checkpoints for Parallel Programs**. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop*, volume 1162 of *Lecture Notes in Computer Science*, pages 140–154. Springer, April 1996.

[12] U. Schwiegelshohn and R. Yahyapour. **Improving First-Come-First-Serve Job Scheduling by Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[13] J. Skovira, W. Chan, H. Zhou, and D. Lifka. **The EASY-LoadLeveler API project**. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, April 1996.

[14] W. Smith, V. Taylor, and I. Foster. **Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance**. In *Proceedings of the 5th Annual Workshop on Job Scheduling Strategies for Parallel Processing*, April 1999. In conjunction with IPPS/SPDP'99, Condado Plaza Hotel & Casino, San Juan, Puerto Rico.

[15] K. Suzaki and D. Walsh. **Implementation of the Combination of Time Sharing and Space Sharing on AP/Linux**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[16] C. Z. Xu and F. C. M. Lau. **Load Balancing in Parallel Computers: Theory and Practice**. Kluwer Academic Publishers, Boston, MA, 1996.

[17] K. K. Yue and D. J. Lilja. **Comparing Processor Allocation Strategies in Multiprogrammed Shared-Memory Multiprocessors**. *Journal of Parallel and Distributed Computing*, 49(2):245–258, March 1998.

[18] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramanian. **Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques**. In *Proceedings of IPDPS 2000*, Cancun, Mexico, May 2000.

Memory Management Techniques for Gang Scheduling*

William Leinberger, George Karypis, and Vipin Kumar

Army High Performance Computing and Research Center, Minneapolis, MN
Department of Computer Science and Engineering, University of Minnesota
(leinberg, karypis, kumar)@cs.umn.edu

Abstract. The addition of time-slicing to space-shared gang scheduling improves the average response time of the jobs in a typical job stream. Recent research has shown that time-slicing is most effective when the jobs admitted for execution fit entirely into physical memory. The question is, how to select and map jobs to make the best use of the available physical memory. Specifically, the achievable degree of multi-programming is limited by the memory requirements, or physical memory pressure, of the admitted jobs. We investigate two techniques for improving the performance of gang scheduling in the presence of memory pressure: 1) a novel backfill approach which improves memory utilization, and 2) an adaptive multi-programming level which balances processor/memory utilization with job response time performance. Our simulations show that these techniques reduce the average wait time and slow-down performance metrics over naive first-come-first-serve methods on a distributed memory parallel system.

1 Introduction

Classical job scheduling strategies for parallel supercomputers have centered around space-sharing or processor sharing methods. A parallel job was allocated a set of processors for exclusive use until it finished executing. Furthermore, the job was allocated a sufficient number of processors so that all the threads could execute simultaneously (or gang'ed) to avoid blocking while attempting to synchronize or communicate with a thread that had been swapped out. The typically poor performance of demand-paged virtual memory systems caused a similar problem when a thread was blocked due to a page fault. This scenario dictated that the entire address space of the executing jobs must be resident in physical memory [2, 4, 10]. Essentially, physical memory must also be gang'ed with the processors allocated to a job. While space-sharing provided high execution rates, it suffered from two major draw-backs. First, space-sharing resulted

* This work was supported by NASA NCC2-5268 and by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 252–261, 2000.
© Springer-Verlag Berlin Heidelberg 2000

in lower processor utilization due to some processors being left idle. These processor "holes" occurred when there were not sufficient processors remaining to execute any of the waiting jobs. Second, jobs had to wait in the input queue until a sufficient number of processors were freed up by a finishing job. In particular, many small jobs (jobs with small processor requirements) or short jobs (jobs with short execution times) may have had to wait for a single large job with a long execution time, which severely impacted their average response time.

Time-slicing on parallel supercomputers allows the processing resources to be shared between competing jobs. Each parallel job is gang-scheduled on its physical processors for a limited time quantum, TQ. At the end of the time quantum, the job is swapped out and the next job is swapped in for its TQ. This improves overall system utilization as processors which are idle during one time quantum may be used during a different time quantum. Gains in response time are also achieved, since the small and short jobs may be time-sliced with larger, longer running jobs. The larger jobs make progress, while the small jobs are not blocked for long periods of time [5, 11].

With time-slicing, a jobs' execution rate is dependent on the number of other jobs mapped to the same processors, or the effective multi-programming level (MPL). Time-slicing systems typically enforce a maximum MPL to control the number of jobs mapped to the same physical processors. For job streams with a high percentage of small or short jobs, increasing the maximum MPL generally increases the benefits of time-slicing. Herein lies the primary impact of memory considerations to time-sliced gang scheduling. The maximum achievable MPL is *limited* by the need to have the entire address space of all admitted jobs resident in physical memory. Therefore, effective time-slicing requires efficient management of physical memory in order to maximize the achievable MPL. Additionally, the current benefits of high processor utilization and job response time must be maintained [1, 12].

Our first contribution is a novel approach for selecting jobs and mapping them to empty slots through a technique which we call *weighted memory balancing*. Job/slot pairs are selected which maximize the memory utilization, resulting in a higher achievable MPL. However, aggressive memory packing can lead to fragmentation in both the memory and the processors, causing delays to jobs with large resource requirements. Therefore, our second contribution is to provide an *adaptive multi-programming level* heuristic which balances the aggressive memory packing of small jobs with the progress requirements of large jobs.

The remainder of this paper is as follows. Section 2 provides an overview of the state-of-the-art in time-sliced gang scheduling (referenced as simple gang scheduling in current literature). We describe our new memory-conscious techniques in Section 3. We also describe their integration into current methods. Section 4 describes a simulation exercise that we used to evaluate our techniques on synthetic job streams. Included in this section is our model for a dual-resource (processor and memory) job stream. Section 5 concludes with a discussion of our work-in-progress.

2 Preliminaries

2.1 System Model

Our system model is a distributed memory parallel system with P processors. Each processor has a local memory of size M. All processors are interconnected tightly enough for efficient execution of parallel applications. We also assume that there is no hardware support for parallel context switching. Time-slicing parallel applications on these types systems is generally conducted using a coarse, or large, time quantum to amortize the cost of context switching between jobs. Finally, we assume that the cost to swap between memory and disk for a context switch or in support of demand paging virtual memory is prohibitively high, even given the coarse time quantum. This model encompasses both the parallel supercomputers, like the IBM SP2 or Intel Paragon, as well as the emerging Beowulf class PC super-clusters. Examples supporting this system model are the IBM ASCI-Blue system [8], and the ParPar cluster system [1].

2.2 Job Selection and Mapping for Gang Scheduling

Given a p-processor job, the gang scheduler must find an empty time slot in which at least p processors are available [9]. While many methods have been investigated for performing this mapping [3], the Distributed Hierarchical Control (DHC) method achieves consistently good performance [6]. We use DHC as our baseline mapping method. The DHC method uses a hierarchy of controllers organized in a buddy-system to map a p-processor parallel job to a processor block of size $2^{\lceil log_2(p) \rceil}$. A controller at level i controls a block of 2^i processors. The parent controller controls the same block plus an adjacent "buddy" block of the same size. DHC maps a p-processor job to the controller at level $i = \lceil log_2(p) \rceil$ which has the lightest load (fewest currently mapped jobs). This results in balancing the load across all processors.

The selection of the next job for mapping is either first-come-first-serve (FCFS), or a re-ordering of the input queue. FCFS suffers from blocking small jobs in the event that the next job in line is too large for any of the open slots. Backfilling has commonly been used in space-shared systems for reducing the effects of head-of-line (HOL) blocking by moving small jobs ahead of large ones. EASY backfilling constrains this re-ordering to selecting jobs which do not interfere with the earliest predicted start-time of the blocked HOL job [7]. This requires that the execution times of waiting jobs and the finishing times of executing jobs must be calculated. In a time-sliced environment, an approximation to the execution time of a job is the predicted time of the job on an idle machine times the multi-programming level [13]. Various backfill job selection policies have been studied such as first-fit which selects the next job in the ready queue for which there is an available slot, and best-fit which selects the largest job in the ready queue for which there is an available slot.

2.3 Gang Scheduling with Memory Considerations

Research into the inclusion of memory considerations to gang scheduling is just beginning. Batat and Feitelson [1] investigated the trade between admitting a job and relying on demand-paging to deal with memory pressure vs queueing the job until sufficient physical memory was available. For the system and job stream tested, it was found to be better to queue the job. Setia, Squillante, and Naik investigated various re-ordering methods for gang scheduling a job trace from the LLNL ASCI supercomputer [12]. One result of this work is that restricting the multi-programming level can impact the response time of interactive jobs as the batch jobs dominate the processing resources. They proposed a second parameter which limited the admission rate of large batch jobs.

3 Memory Management Techniques for Gang Scheduling

As a baseline, we use the DHC approach to gang scheduling. However, we integrate the job selection and mapping processes of DHC so that we can select a job/slot pair based on memory considerations. We provide an intelligent job/slot selection algorithm which is based on balancing the memory usage among all processors, much like the DHC balances the load across all processors. This is described below in Section 3.1. This job/slot selection algorithm is based on EASY backfilling, which is subject to blocking the large jobs [13]. We also provide an adaptive multi-programming level which is used to control the aggressiveness of the intelligent backfilling. This adaptive MPL is described below in Section 3.2.

3.1 Memory Balancing

Mapping a job based on processor loading alone can lead to a system state where the memory usage is very high on some processors but very low on others. This makes it harder to find a contiguous block of processors with sufficient memory for mapping the next job. We refer to this condition as *pre-mature memory depletion*. Figure 1 shows a distributed memory system with P=8, M=10, and MPL=3. Consider mapping a job, J_6, with processor and memory requirements $J_6^P = 1$ and $J_6^M = 4$ respectively. Figure 1 (a) depicts the result of a level 0 controller mapping J_6 to processor/memory P_0/M_0, in time quantum slot TQ_1, using a first-fit (FF) approach. This results in depleting most of M_0 on P_0, making it difficult for this controller to map additional jobs in the remaining time slot, TQ_2. The parent controller, which maps jobs with a 2-processor requirement to the buddy-pair P_0 and P_1 will also be limited. Continuing, all ancestor controllers will be limited by this mapping, to jobs with $J_i^M \leq 1$.

An alternative mapping is depicted in Figure 1 (b). Here, J_6 is mapped to P_4, leaving more memory available for the ancestral controllers to place future jobs. Processor 4 was selected because it had the lowest load (one job) *and* left the most memory available for other jobs. One heuristic for achieving this is as follows. Map the job to the controller which results in balancing the memory

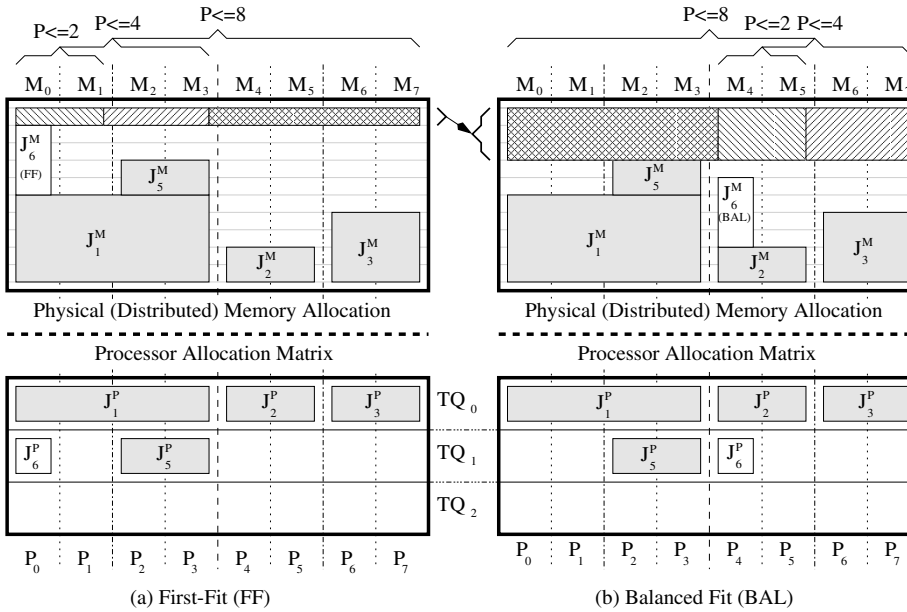


Fig. 1. Avoid premature memory depletion, (a), through memory balancing, (b). Shaded regions depict memory used by currently executing jobs, while hashed regions depict memory free for allocation to future jobs by ancestral controllers.

utilization across all the processors. Let M_i^T be the total memory usage on processor P_i. That is, $M_i^T = \sum_{J_j \in P_i} J_j^M$. We define the memory balance measure as $BAL = Max(M_i^T)/Avg(M_i^T), 0 \leq i < P$. Note that $BAL \geq 1$, and $BAL = 1$ indicates that all physical memory is fully utilized.

This notion can be further refined by noting that mapping a job to a controller on level i directly affects the memory on the 2^i processors managed by that controller, so we first measure the balance across these processors. This local balance score is weighted by the probability that a job with processor requirement $2^{i-1} < J_j^P \leq 2^i$ arrives in the future. Essentially, the weighted balance score measures the ability of the controller to meet the memory requirements of future jobs on its processors. Continuing, measure the balance across the 2^{i+1} processors controlled by the parent of this controller, and so on, until we measure the balance across the entire range of processors in the system. As we progress up the levels of controllers, the balance measured at each controller is weighted by the probability of needing a slot of the size managed by that controller. The total score is the average of the weighted scores at each level. The processor requirement probability distributions are derived by keeping track of the sizes of jobs which have been previously scheduled. During a scheduling epoch, each job in the ready queue is scored against each possible slot. The job/slot pair with the best score is selected for admission, subject to the maximum MPL constraint.

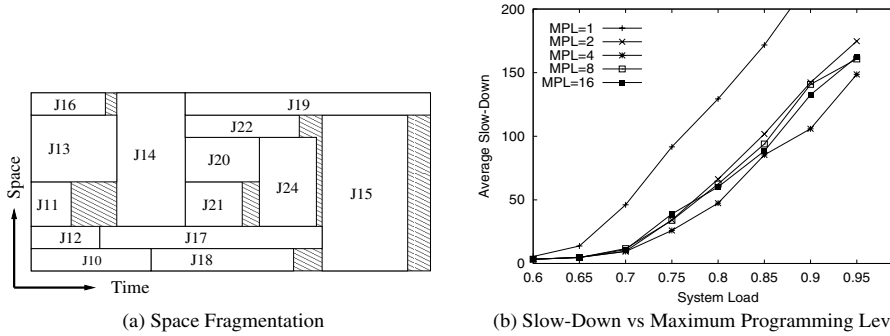


Fig. 2. (a) Aggressive backfilling causes space fragmentation, blocking large jobs. (b) Increasing the MPL beyond a "natural" level leads to over-aggressive backfilling.

3.2 Adaptive Multi-programming Level

Aggressive backfilling methods move smaller jobs ahead of larger blocked jobs in an effort to improve average job response time. The backfill jobs may not interfere with the predicted start time for the job blocked at head of the Ready Queue (RQ). However, the job which is *next-in-line* in the queue may be delayed severely by the backfilling due to *space fragmentation*. Consider the system state depicted in Figure 2 (a). The jobs are numbered according to their arrival, with J_{10} arriving before J_{11}, etc.. At some point, job J_{14} was delayed at the head of the queue, with J_{15} right behind it. J_{16} and J_{17} are backfilled, since neither interferes with the earliest start time for J_{14}. However, J_{17} inadvertently delays J_{15}. This space fragmentation effect compounds against the third, fourth, etc., jobs waiting at the head of the ready queue. Although the small jobs are moved ahead, the large jobs may be delayed a disproportionately long time. Note that the space axis in Figure 2 (a) may represent processor usage or memory usage as fragmentation can occur in either resource. DHC naturally avoids processor fragmentation due to its use of a buddy system for mapping jobs to processors. However, space fragmentation in the memory on each processor can still occur, delaying jobs with relatively large memory requirements.

Figure 2 (b) depicts the average slow-down performance of a workload in which the average per-processor memory requirement is 25% of the available physical memory. For MPL ≤ 4, the performance increases with increasing MPL. However for MPL > 4, the performance decreases, due to over-aggressive backfilling. We developed a heuristic to adaptively adjust the maximum MPL, based on the *natural* level dictated by the waiting jobs. The natural level is the total memory per processor, M, divided by the average per-processor memory requirement of the jobs waiting in the queue. If the backfilling is temporarily achieving a multi-programming level above this natural level, then a lot of jobs with small memory requirements are being selected in favor of the jobs with

larger memory requirements. Periodically, the maximum MPL is re-calculated as $\lceil M/Avg(J_i^M) \rceil, J_i \in RQ$.

4 Experimental Results

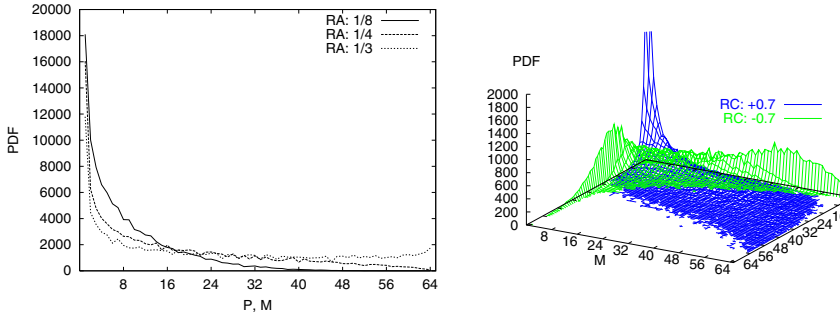
In this section, we describe the simulation methods used to evaluate the new techniques. Our dual-resource workload model is described in Section 4.1, and our simulation results are presented in Section 4.2.

4.1 Workload Model

Past research on workload models has focused on a single resource, processors [3]. The results of these efforts generally provide that processor requirements follow a hyperexponential distribution, with many strong discrete components at powers of two and squares of integers. Recent efforts is characterizing memory usage show that memory and processor requirements are weakly correlated [4, 1]. Also, the size of the memory requirements, while high, still allow for a low degree of multi-programming in physical memory [12]. We generalize this conclusion with the following dual-resource workload model. First, the probability distributions for the processor and memory resource requirements are generated with a specific mean, R_A, and variance, R_V. Example distributions are depicted in Figure 3 (a). The requirements for a given job are then drawn from the respective distributions in such a way as to create a job stream in which the processor and memory requirements are correlated as specified by a resource correlation parameter, R_C. Histograms for two values of R_C are depicted in Figure 3 (b). The X and Y axis represent the various values of P and M, while the Z axis is the number of times that combination of P and M were generated in job. The execution times for jobs are drawn from a hyperexponential distribution as well. The inter-arrival times are exponential, and system load is adjusted by changing the inter-arrival rate. For this exercise, we omit the strong discrete components, as it is not well understood which sizes are important for the memory distribution, or how they correlate to discrete sizes for the processor distribution.

4.2 Simulation Results

We implemented the DHC time-sliced gang scheduler on a simulated system, with three different job-selection/mapping algorithms. The first job selection algorithm is the first-come-first-serve (FCFS), which takes jobs from the head of the queue and places them onto the least loaded slot of the appropriate size, with sufficient physical memory. Second, the FCFS was modified to include EASY backfilling (EASY). Finally, the first-fit job selection method used by EASY was replaced by the weighted memory balancing job selection method described in Section 3.1 (WBAL). The EASY and WBAL algorithms were also simulated with the adaptive MPL as described in Section 3.2, and are denoted as EASY/AM and WBAL/AM respectively. We assume perfect knowledge for



(a) Resource Probability Distributions (b) Resource Correlation Histograms

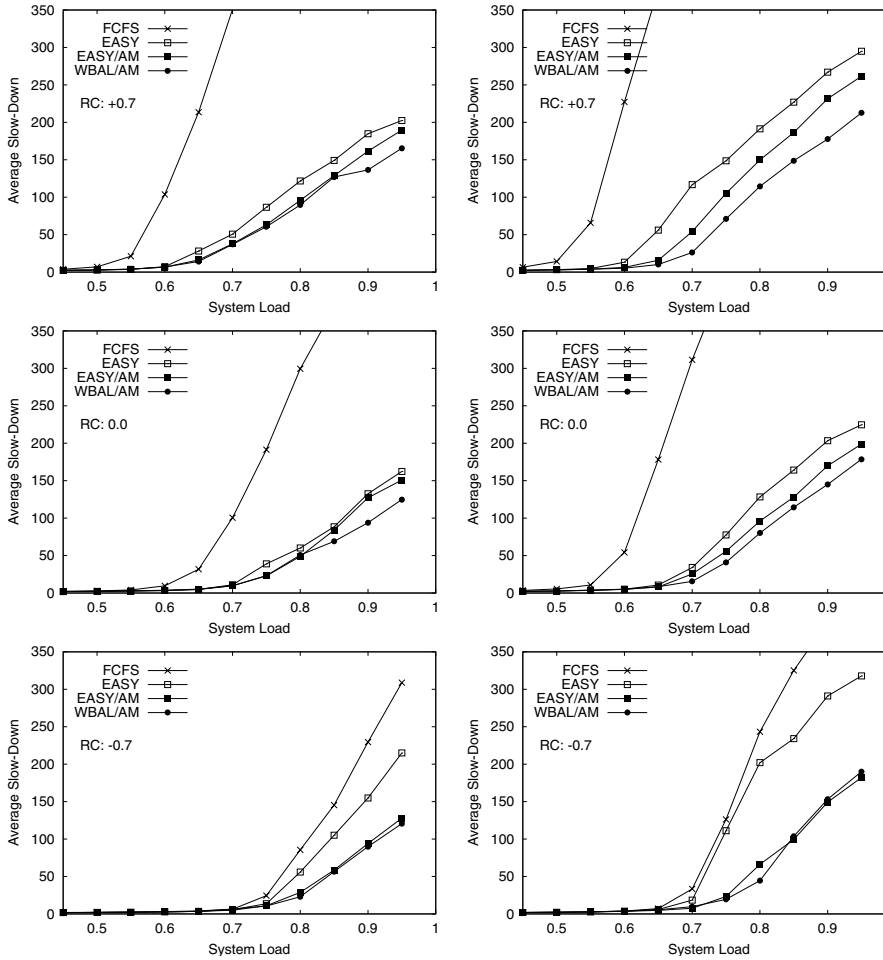
Fig. 3. Dual resource workload model. Processor and memory requirements are drawn from single resource distributions, (a), at various correlation levels, (b).

resource requirements and execution times. Relaxing this assumption is the subject of our work-in-progress, discussed briefly in Section 5.

The simulated parallel system used P=64 and M=64. The algorithms were evaluated on the basis of the average slow-down performance metric. The slow-down metric is the ratio of the execution time on the loaded machine (wait time plus reduced execution rate) to the execution time on an idle machine (no waiting, full execution rate). We used a single distribution for the processor requirements with the $R_A^P = 1/8$, or roughly 8 of the 64 available processors. We used two different memory requirement distributions, $R_A^M = 1/4$, and $R_A^M = 1/3$. Results are also reported for three different resource correlation values, $R_C = +0.7$, $R_C = 0.0$, and $R_C = -0.7$. The average slow-down performance results are depicted in Figure 4. Figure 4 (a) depicts results for $R_A^M = 1/4$ over all three values for R_C and (b) depicts similar results for $R_A^M = 1/3$. Overall, the WBAL/AM consistently performs as good or better than the EASY/AM and EASY. Additionally, EASY/AM performs as good or better than EASY.

At lower memory pressure, Figure 4 (a), the backfill based algorithms perform about the same, with WBAL/AM slightly better than EASY/AM and EASY. EASY/AM and WBAL/AM perform well due to the fact that the multiprogramming level is naturally large, as jobs have low per-processor memory requirements. The AM heuristic prevents EASY/AM and WBAL/AM from aggressive backfilling, thus avoiding the memory space fragmentation produced by EASY. When the processor and memory requirements are negatively correlated, EASY/AM and WBAL/AM perform much better than EASY. In this case, the many small processor jobs generally have high memory requirements. The combination of un-restrained MPL and the first-fit mapping used by EASY results in pre-mature memory depletion.

At higher memory pressure, Figure 4 (b), and higher resource correlation, the improved packing efficiency produced by WBAL and moderated by AM results in WBAL/AM achieving better performance than the EASY backfill variants.



(a) Average Memory Requirement: 1/4

(b) Average Memory Requirement: 1/3

Fig. 4. Average Slow-Down Results

This is significant as these workloads correspond most closely to the findings of the studies on memory requirements for scientific workloads, described earlier ($R_C = 0.7$ is considered weakly correlated). As the resource correlation goes negative, the performance of EASY/AM and WBAL/AM increases over EASY, as a result of the AM heuristic preventing overly aggressive backfilling.

5 Summary and Future Work

The combination of the weighted memory balancing and the adaptive multi-programming level heuristics produced a job selection and mapping algorithm,

WBAL/AM, which consistently outperformed other backfill methods. However, backfill methods require apriori knowledge of resource requirements and execution times. Our future work is aimed at over-coming this limitation. Basically, jobs are selected and mapped using information that may initially have errors. Once jobs have executed for a few time slices, resource requirement information is improved. In the event that memory becomes over-subscribed this can be used to decide which jobs to *swap out* of memory to disk.

References

[1] A. Batat and D.G. Feitelson. Gang scheduling with memory considerations. In *Proceedings of the ISPDP '2000*. IEEE Computer Society, May 2000.

[2] D.C. Burger, R.S. Hyder, B.P. Miller, and D.A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Supercomputing '94*, 1994.

[3] D.G. Feitelson. Packing schemes for gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162, pages 65–88. Springer-Verlag, New York, 1996. LNCS.

[4] D.G. Feitelson. Memory usage in the lanl cm-5 workload. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291, pages 78–94. Springer-Verlag, New York, 1997. LNCS.

[5] D.G. Feitelson and M.A. Jette. Improved utilization and responsiveness with gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291. Springer-Verlag, New York, 1997. LNCS.

[6] D.G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *J. of Parallel and Distr. Comp.*, 1996.

[7] D.G. Feitelson and A.M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Proceedings of the IPPS/SPDP 1998*, pages 542–546. IEEE Computer Society, 1998.

[8] H. Franke, J. Jann, J.E. Moreira, and P. Pattnaik. An evaluation of parallel job scheduling for asci blue-pacific. In *Supercomputing '99*, November 1999.

[9] J.K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd Intl. Conf. Distributed Computing Systems*, pages 22–30, Oct 1982.

[10] V.G.J. Peris, M.S. Squillante, and V.K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 5–18, 1994.

[11] U. Schwiegelshohn and R. Yahyapour. Improving first-come-first-serve job scheduling by gang scheduling. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459. Springer-Verlag, New York, March 1998. LNCS.

[12] S. Setia, M.S. Squillante, and V.K. Naik. The impact of job memory requirements on gang-scheduling performance. Technical Report RC 21373, IBM T.J.Watson Research Center, March 1999.

[13] Y. Zhang, H. Franke, J.E. Moreira, and A. Sivasubramaniam. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. Technical Report RC 21579, IBM T.J.Watson Research Center, October 1999.

Exploiting Knowledge of Temporal Behaviour in Parallel Programs for Improving Distributed Mapping*

Concepció Roig[1], Ana Ripoll[2], Miquel A. Senar[2], Fernando Guirado[1], and Emilio Luque[2]

[1] Universitat de Lleida, , Dept. of CS
Jaume II 69, 25001 Lleida, Spain
roig@eup.udl.es, fernando@eup.udl.es
[2] Universitat Autònoma de Barcelona, Dept. of CS
08193 Bellaterra, Barcelona, Spain
a.ripoll@cc.uab.es, m.a.senar@cc.uab.es, e.luque@cc.uab.es

Abstract. In the distributed processing area, mapping and scheduling are very important issues in order to exploit the gain from parallelization. The generation of efficient static mapping techniques implies a previous modelling phase of the parallel application as a task graph, which properly reflects its temporal behaviour. In this paper we use a new model, the Temporal Task Interaction Graph (TTIG), which explicitly captures the temporal behaviour of program tasks; and we evaluate the advantages that derive from the use of the TTIG model in task allocation. Experimentation was performed in a current PVM environment, for a set of synthetic graphs which exhibit different ratios of computation/communication cost (coarse-grain, medium-grain). The execution times when these programs were mapped using the information contained in the TTIG model, were compared with the times obtained using the two following mapping alternatives: (a) PVM default scheme and, (b) mapping strategy based on the classical model TIG (Task Interaction Graph). The results confirm that with the TTIG model, better assignments are obtained, providing improvements of up to 49% compared with the PVM assignments and up to 30% compared with TIG assignments.

1 Introduction

Parallel programming based on message-passing presents the programmers with daunting problems when attempting to achieve efficient execution. The parallel solution of a problem evolves across three phases. First, we devise a parallel algorithm solving the problem in hand. Then, an interacting task network (a task graph) implementing the algorithm on the available computational model is designed. Finally, the tasks are mapped onto the target architecture. The task graph design and physical mapping must exploit the potential parallelism detected to build an efficient implementation on the target machine. These two

* This work was supported by the CICYT under contract TIC98-0433

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 262–271, 2000.
© Springer-Verlag Berlin Heidelberg 2000

phases, although often solved separately in the solutions proposed in the literature, are closely related. The chosen structure for the task graph strongly affects the efficiency with which we can address the mapping and scheduling problems.

Most of the approaches and standards recently proposed for programming parallel machines provide explicit parallel models in which programmers are aware of the parallel execution of the programs, and are asked to define the task graph. These models give the programmer control over both the decomposition of activities between tasks and the management of communication/synchronization between the parallel tasks. Libraries provide interacting primitives that can be called from within sequential codes (usually C and Fortran), and facilities to define the task graph structure. This is the case of PVM and MPI.

Two task graph models have been extensively used in the areas of mapping and scheduling. The **Task Precedence Graph (TPG)** [1] and the **Task Interaction Graph (TIG)** [2]. TPG is a directed graph where the nodes and directed edges represent the tasks and tasks precedence constraints respectively. This TPG model supposes that the tasks can interact only at the beginning and at the end of their execution. On the other hand, TIG is an undirected graph where two tasks communicate if there is an edge between the nodes. This TIG model allows us to model arbitrary interactions between the parallel tasks, and communications can take place at any point inside them.

In both graph models, weights are associated to nodes and edges, representing computation and communication times respectively. The choice of the graph model depends on the structure of the parallel activity of the application and on the abstraction level we are interested in. The TPG approach is particularly effective for many scientific applications where interactions between tasks take place only at the beginning and at the end of their execution. On the other hand, distributed processing applications where the executing tasks are required to communicate during their lifetime rather than just at the initiation and at the end, are successfully modelled by the TIG. However, as the TIG model does not include any information about task precedences, we cannot take advantage of the potential parallelism between tasks. For this reason most of the authors prefer to assume that all tasks may run in parallel and the requirement made is to minimize the number of tasks mapped onto the same processor [7][8].

Accordingly, we have proposed a new task graph model called **Temporal Task Interaction Graph** (TTIG), which in addition to the weights considered in the two previous models, explicitly captures the potential degree of parallel execution between adjacent tasks [3]. With the TTIG, a more realistic way of representing the behaviour of applications with explicit parallelism is provided.

The aim of this paper is to evaluate the advantages that derive from the knowledge of task behaviour by modelling the parallel program with the TTIG. The effectiveness of this program model has been proved in a real PVM message-passing environment for a set of synthetic programs, and the results obtained confirm that the TTIG is a good model, and allows us to solve the mapping and scheduling of arbitrary parallel computations in an effective way.

The remaining sections are organized as follows. Section 2 summarizes the new TTIG model and emphasizes its main characteristics. Section 3 presents the experimental results obtained for a set of synthetic graphs on a PVM platform, and section 4 outlines the main contributions.

2 The Parallel Program Model

In a message-passing computation model, a parallel application is a set of sequential processes (tasks) communicating by calling library routines to send and receive messages. Communications can be point to point i.e. involving a message transfer from one named task to another, or collective, performing global interactions between a (sub)set of tasks in the program. A task is considered as an indivisible unit of computation to be scheduled on one processor, and the grain size of tasks is determined by the programmer. In this model, send operations are supposed to be non-blocking, that is, we assume that the sending task can continue its execution after sending a message. On the contrary, receive operations are blocking and the receiving task can only continue its execution when the message has been received.

In principle, a task can be seen as a set of computation phases with the necessary communication to provide and/or to obtain the data for the next computation phase. A task graph that shows the interactions between different computation phases of each task can have an arbitrary number of nodes, each one executing a distinct sequential process. In this graph, each directed edge represents a send operation between tasks and it is labeled with a number representing the communication cost (the cost involved in exchanging data). The numbers included inside each node represent the cost of each of its computation phases. This graph is called **Temporal Flow Graph (TFG)** and reflects precedence relationships between program tasks. This TFG graph can be explicitly specified by the programmer, deduced automatically by the compiler or refined by dynamic monitoring of different executions of the application [4] [5].

Fig. 1(a) shows an example of TFG for a parallel program with 5 tasks {T0,..,T4}; in this graph, dashed lines are computation phases inside each task that have to be executed sequentially; and continuous lines represent the communications established between neighbouring tasks. The computation phases and corresponding computation cost can be seen for every task (i.e. task T3 has two computation phases with a computation cost of 30 each), and the communications with its cost (i.e. task T3 has to establish three communications: two *sending* operations to tasks T0 and T4 with a cost of 12 and 2 respectively and a *receiving* operation from task T0 with cost 9).

Starting from the TFG of a parallel program, it is possible to obtain some information about the temporal behaviour of program tasks. Accordingly, we define for each pair of adjacent tasks a new parameter called *degree of parallelism*. Thus, for two communicating tasks with message transferences from Ti to Tj in the TFG graph, this parameter is defined as the percentage of Tj execution time that can be carried out in parallel with Ti. In the same way, if there are

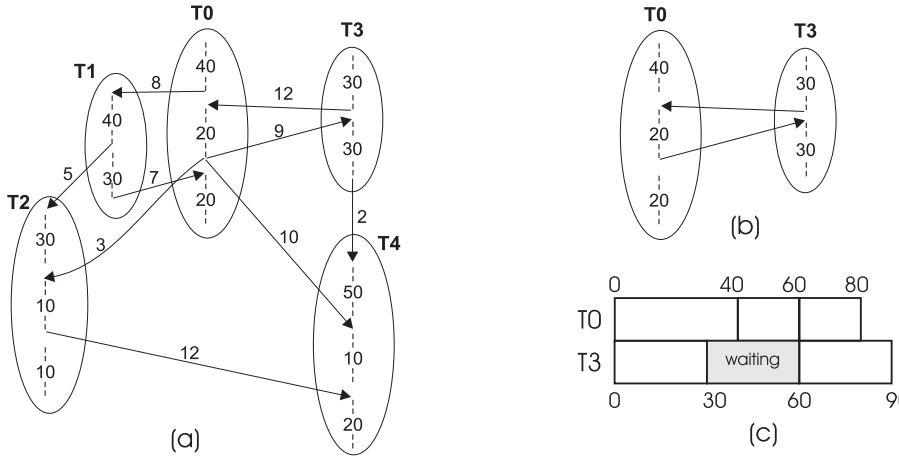


Fig. 1. (a) TFG graph, (b) subgraph of tasks T0 and T3 and (c) Execution simulation

messages sent from Tj to Ti the degree of parallelism is defined with respect to Ti execution time. The degree of parallelism is represented as a normalized index belonging to the [0,1] interval

This degree of parallelism is obtained for each pair of adjacent tasks in the TFG assuming that they are isolated from the rest of the graph and without considering the cost involved in communications. For instance, for tasks T0 and T3 of TFG graph in Fig. 1(a) the degree of parallelism is obtained as following: we simulate the execution of the corresponding subgraph (Fig. 1(b)) and we evaluate the time during which these tasks are executing concurrently; in this case, as it can be seen in Fig. 1(c) we obtain 50 time units of parallel execution, so the degree of parallelism from task T0 to task T3 is 50/60=0.83 and that corresponding from task T3 to T0 is 50/80=0.6. A more detailed description about the obtaining of this parameter can be found in [3].

With this new parameter, the new graph model called **Temporal Task Interaction Graph (TTIG)** is build. The TTIG is a directed graph, where nodes represent the tasks of the parallel program with its associated execution time (this is the sum of each computation cost for every phase inside the task) and the edges indicate directed communications between tasks with two associated parameters: (a) the global communication cost (this is the sum of each communication cost for every transference established between the two adjacent tasks in the direction of the edge) and (b) the degree of parallelism existing between them. Fig. 2 shows the TTIG graph obtained for the TFG graph of Fig. 1(a). In this graph, for example, task T3 has an execution time of 60 resulting from the sum of two computation phases with a cost of 30; and the communication costs from T0 to T3 and from T3 to T0 are 9 and 12 respectively, resulting from

the sum of the communication costs involved in message transferences in both directions.

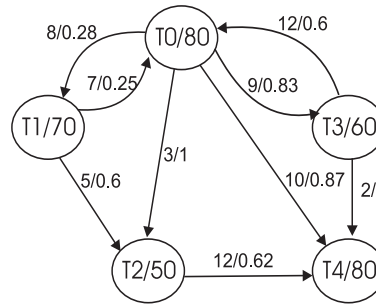


Fig. 2. TTIG graph

The TTIG model integrates the two classical TPG and TIG models. Thus, if two tasks have a precedence constraint as happens in the TPG graph, this is reflected in the TTIG with a maximum degree of parallelism equal to 0 (this is the case of tasks T3 and T4 in the TTIG of Fig. 2). On the other hand, a maximum degree of parallelism of 1 in the TTIG states that a task can execute concurrently all the time with its adjacent, as is considered in the TIG. With other values for degree of parallelism, any other situation can be reflected in the TTIG graph, so this is a good model that allows us to represent the temporal behaviour of a parallel program with any pattern of interaction between tasks.

In order to automate the process for obtaining the TTIG graph, a tool has been designed and implemented. This tool has a graphical user interface which provides an easy-to-use interactive environment for introducing the parametrized TFG graph of a message-passing program, and it automatically generates the corresponding TTIG graph with the associated parameters.

3 Experimental Study on a PVM Platform

In this section we analyze the effectiveness of the TTIG, assuming a static approach, to solve the mapping problem. That is, we assume a knowledge of application behaviour before starting program execution, and the criterion to optimize is the minimization of the completion time of the program execution. We have generated a set of C+PVM synthetic programs and we have compared the global execution time obtained when these programs were mapped using the information contained in the TTIG model with the time obtained using the two following allocation alternatives: (a) PVM default scheme which is based on a round-robin policy, and (b) TIG strategy based on the criteria of load balancing and minimization of communication cost. We refer to these three different allocations as TTIG mapping, PVM mapping and TIG mapping, respectively.

The programs used in this experimentation (pr_1,..,pr_7) were randomly generated and had a number of tasks ranging from 6 to 10. Each task graph was a PVM application and consisted of a set of computation phases (based on integer addition loops) with communication primitives (with both sending and receiving operations). All tasks had a uniform execution time (8 seconds), in order to highlight the contribution of the degree of parallelism in achieving an efficient distribution of tasks.

The experiments were carried out varying the computation/communication ratio of the programs. Firstly, we used *coarse-grain* programs where messages consisted of one integer. In this case, the size of messages is very small and the time incurred in communication is negligible. The same set of programs was used exhibiting a *medium-grain*, where the size of messages ranged between 25.000 and 75.000 integers. In this case, the time incurred by communications ranged between 1 and 3 seconds. Overall, in *medium-grain* programs the ratio between computation and communication ranged between 1,1 and 1,5.

The number of integer additions and the size of messages was fixed according to the results obtained by the evaluation of the performance of our system having it dedicated to the execution of our application. In this case, the addition of 100.000.000 integers takes approximately one second; moreover, a ping-pong experiment, where series of messages are sent between two tasks reported a cost of approximately one second for sending a message of 25.000 integers.

The experiments were conducted on 4 Linux (kernel V. 2.0.36) machines running PVM 3.4. Each machine was based on a Pentium II at 350MHz, with 128Mbytes of RAM and 512Kbytes of cache each. The interconnection network was a 100Mbs Fast Ethernet. Each program was executed on this platform using three different allocations of tasks, based on the following mapping strategies:

(a) *PVM mapping.* PVM default allocation scheme assigns the tasks to the available hosts using a round-robin policy [6]. Once a task is started, it runs on the assigned host until completion, i.e., the task is statically allocated. Obviously, a round-robin allocation may be very sensitive to the order under which tasks are created. Therefore, in our experiments several task allocations of the graphs were tried initially in order to check which task creation order obtained the best performance. The results reported below have been computed using the PVM allocation with the task ordering that obtained the best performance, i.e. these allocations may be considered as the allocations obtained by a highly experienced PVM programmer.

(b) *TIG mapping.* In this case, the same set of programs were modelled as a TIG graph, and the mappings were done using the heuristic CREMA [8], which provides excellent results compared with other mapping algorithms of TIG's existing in the literature [9]; this is a mapping heuristic based on the *minimax* cost function. Under the *minimax* criterion, the goal is to minimize the maximum *Processor Work Load* (PWL), where the PWL for each processor is defined as the total cost due to the computation and communication cost of all the tasks mapped to it (PWL = computation cost + communication cost).

(c) *TTIG mapping.* The TTIG mapping was carried out basically taking into account the degree of parallelism between tasks. Then, nodes of the graph were traversed from top to bottom in order to determine which of tasks were the more dependent i.e. almost unable to execute in parallel, and which of the adjacent tasks were less dependent i.e. capable of doing the most part of their execution concurrently. The allocation was carried out with the following criteria: tasks with a degree of parallelism between 0 and 0.3 were assigned to the same processor and the tasks with a degree of parallelism between 0.7 and 1 were mapped to different processors. For tasks with an intermediate value of degree of parallelism, we used the minimax criterion mentioned above.

Different runs on the same program graph generally produced slightly different final execution times. Hence, average-case results are reported for sets of ten runs of all PVM, TIG and TTIG mappings.

Table 1 and Table 2 contain the average execution times, in seconds, obtained by PVM, TIG and TTIG mappings with *coarse-grain* programs and *medium-grain* programs respectively. It can be seen in both tables that the use of the TTIG model yields significant improvements in the global execution times with respect to PVM and TIG mappings. The execution times obtained for *medium-grain* programs (Table 2) are higher than these obtained for *coarse-grain* programs (Table 1) due to the fact that the transference of a significant amount of communication between tasks implies an additional cost in time that has to be added to the cost of computation phases inside the tasks. Moreover, the times obtained for TIG mappings are in general slightly better than PVM mappings because, with the heuristic CREMA, the allocation strategy is based on the structure of the application graph; however, for some of these structures the use of this kind of strategy that focuses on achieving load balancing and minimization of communication cost, brings assignments that separate the more dependent tasks to different processors and allocate very concurrent tasks to the same processor, preventing a global concurrent execution in the system and yielding a worse execution time (see pr_4, pr_7). This kind of problem does not appear in TTIG mapping because the allocation strategy is based on the knowledge of the ability of concurrent execution for adjacent tasks in the application graph.

In Fig. 3 the percentage of gain in the execution time obtained with the TTIG mapping over the PVM mapping is shown graphically. These values have been computed as $100*(T_{PVM}$ - $T_{TTIG})/T_{PVM}$ where T_{PVM} is the average execution time using the PVM mappings and T_{TTIG} is the average execution time using the TTIG mappings. Equally, Fig. 4 shows the percentage of gain in the execution time obtained with TTIG mapping over TIG mapping.

With the analysis of the results shown in Fig. 3, it can be seen that by using the TTIG model we can obtain significant improvements in the global execution time of parallel programs, which can be up to 49% better when compared with the time obtained using the PVM allocation. More specifically, we obtained significant gains in performance when the number of processors was 2 or 3, but

Table 1. Average execution times, in seconds, for coarse-grain programs

	Coarse-grain								
	2 proc.			3 proc.			4 proc.		
	PVM	TIG	TTIG	PVM	TIG	TTIG	PVM	TIG	TTIG
pr_1	31.7	18.6	16.8	25.3	17.3	16.1	16.1	17.5	15.8
pr_2	51	54.2	40.7	42.1	40.9	34	46.5	43	34
pr_3	51	49	41	44	36	33	43	40.5	33
pr_4	41	45.3	37.1	35	44.7	34	34	40.4	34
pr_5	47.8	48	36.5	41.8	43	36.5	38.1	41.5	36.5
pr_6	42	40	33.4	40.6	44	33	36.5	34.6	31.5
pr_7	52.16	57.3	41.2	46.5	55.4	38.8	36.7	43.3	32

execution times were nearly the same for both mappings with 4 processors. This fact can be explained because in most cases the degree of parallelism between tasks is the factor that principally influences the final execution time of the program; but when 4 processors were used, the effect of dependencies is also insignificant because each task is immediately ready to run once it has received its input messages because, as the examined graphs have a low number of tasks, very few of them are executed concurrently in each processor.

Table 2. Average execution times, in seconds, for medium-grain programs

	Medium-grain								
	2 proc.			3 proc.			4 proc.		
	PVM	TIG	TTIG	PVM	TIG	TTIG	PVM	TIG	TTIG
pr_1	41	22.5	21	36	25.9	20	25.6	24	20
pr_2	54	50.5	47	50	43.4	39	42	41.5	38
pr_3	90.8	82.4	76.7	84.3	67	65	44.3	45	41
pr_4	63	59.7	49	43	50.6	39	41	43	39
pr_5	54	52.8	47	59	56.5	47	55.3	53.3	47
pr_6	51	45.7	38	51	44.1	34	39.8	38	33
pr_7	67	81.5	62	60	77.1	54	61.3	73.9	54

It can also be noticed that the gains obtained with TTIG mappings over PVM mappings were not uniform, and there are big differences among the programs; this is a reasonable result because PVM does not take into account any information related to the application graph in mapping decisions and the gain that we obtained is in some way aleatory. Extreme cases are reported in Fig. 3 for pr_1 and pr_4 programs; the graph of pr_1 application has some tasks with a 0 degree of parallelism which are mapped to different processors, and some other tasks with an associated degree of parallelism equal to 1 that are mapped to the same processor, when using the round-robin policy, bringing about a se-

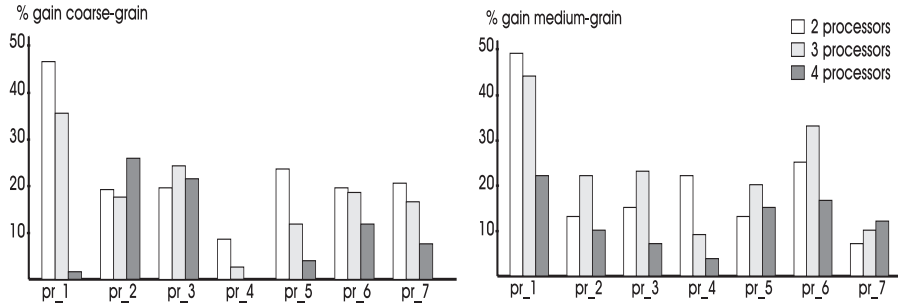


Fig. 3. Percentage of gain of TTIG mapping over PVM mapping.

quential execution of the tasks of the parallel program and, as a consequence, a poor performance. On the other hand, the application graph of the program pr_4 shows little differences in the values of degree of parallelism between tasks; in this case the round-robin assignment based on the spawn ordering had been enough to achieve good performance.

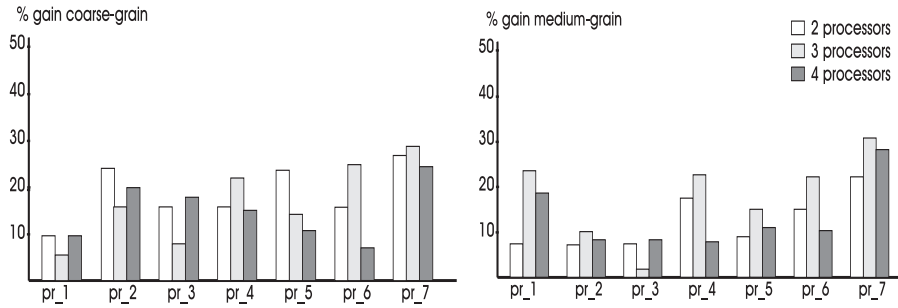


Fig. 4. Percentage of gain of TTIG mapping over TIG mapping.

The use of the TTIG model also yields positive improvement compared with the TIG mapping as is reflected in Fig. 4, and in general, the gain obtained using TTIG mapping is more uniform, when compared with the TIG mapping, than PVM mapping, due to the fact that the TIG mapping is carried out under a criterion based on the structure of the application graph.

Moreover, with the analysis of the results reported in Fig. 4, it can also be seen that in many cases (see pr_2, pr_3, pr_5, pr_6), the percentage of gain of TTIG mapping over TIG is slightly better for coarse-grain programs than medium grain, the reason is that in these programs, in addition to the degree of parallelism, the communication costs also bring about some strong dependencies, the later case is effectively detected by the heuristic CREMA and the tasks involved are properly assigned, yielding better results for global execution time.

4 Conclusions

This work has investigated the advantages of using a new model, the Temporal Task Interaction Graph (TTIG), for representing message-passing parallel programs. The TTIG graph allows us to represent in a more realistic way the behaviour of parallel computations with an unrestricted control and interaction pattern. The degree of parallelism included in the model allows us to solve mapping and scheduling for arbitrary parallel computations in an effective way.

The effectiveness of the TTIG has been established through a mapping experimentation process on a real PVM environment. The programs under experimentation exhibited different ratios of computation/communication cost and the mappings were carried out in three ways: (a) with the PVM default assignment, (b) with a TIG based strategy and (c) based on TTIG characteristics. The results show that using the degree of parallelism as the first decision parameter in mapping is determinant for obtaining significant improvement in global execution time.

References

1. Kwok Y-K. and Ahmad I.: Benchmarking the Task Graph Scheduling Algorithms. Proc. 12th. Int'l Parallel Processing Symp., pp. 531-537, Apr. 1998.
2. Sadayappan P., Ercal R. and Ramanujam J.: Cluster Partitioning Approaches to Mapping Parallel Programs onto Hypercube. Parallel Computing 13, pp. 1-16, 1990.
3. Roig C., Ripoll A., Senar M. A., Guirado F. and Luque E.: Modelling Message-Passing Programs for Static Mapping. Proc. 8th. Euromicro Workshop on Parallel and Distributed Processing. pp. 229-236 Jan. 2000.
4. Fahringer T.: Compile-Time Estimation of Communication Costs for Data Parallel Programs. J. Parallel and Distributed Computing, vol 39, pp. 46-65, 1996.
5. Gupta M. and Banerjee P.: Compile-Time Estimation of Communication Costs on Multicomputers. Proc. Sixth Int. Parallel Processing Symp. Mar. 1992.
6. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press. Cambridge, Massachusets.
7. Hui Ch. and Chanson S.: Allocating Task Interaction Graph to Processors in Heterogeneous Networks. IEEE Trans. on Parallel and Distributed Systems. vol 8, n. 9, pp 908-925 Sep. 1997.
8. Senar M. A., Ripoll A., Cortés A. and Luque E.: Clustering and Reassignment-base Mapping Strategy for Message-Passing Architectures. Int. Par. Proc Symp&Sym. On Par. Dist. Proc. (IPPS/SPDP 98) 415-421. IEEE CS Press USA, 1998.
9. Senar M. A., Ripoll A., Cortés A. and Luque E.: Performance Comparison of Strategies for Static Mapping of Parallel Programs. High Performance Computing and Networking (HPCN97) Lecture Notes in Computer Science 225, 575-587. Springer. Germay, 1997.

Preemptive Task Scheduling for Distributed Systems*

Andrei Rădulescu and Arjan J.C. van Gemund

Delft University of Technology, The Netherlands

Abstract. Task scheduling in a preemptive runtime environment has potential advantages over the non-preemptive case such as better processor utilization and more flexibility when scheduling tasks. Furthermore, preemptive approaches may need less runtime support (e.g. no task ordering required). In contrast to the non-preemptive case, preemptive task scheduling in a distributed system has not received much attention. In this paper we present a low-cost algorithm, called the Preemptive Task Scheduling algorithm (PTS), which is intended for compile-time scheduling of coarse-grain problems in a preemptive distributed-memory system. We show that PTS combines the low-cost of the algorithms for the non-preemptive case with a simpler runtime support, while the output performance is still at a level comparable to the non-preemptive schedules.

1 Introduction

Compile-time scheduling for distributed-systems has recently received considerable attention in the context of non-preemptive environments in which tasks are scheduled to run uninterrupted until completion [1, 3, 6, 8, 9]. In particular it has been shown that there exist several scheduling algorithms (e.g., DSC [9], HLFET [1], FCP [6]) that pair good performance with low cost [6]. Efficient algorithms in the non-preemptive case are typically focused on scheduling first the most "important" tasks (i.e., tasks where delaying their execution will cause a longer completion time).

A preemptive scheme may be more attractive, as a less important task can run while waiting an important task to become ready. We can distinguish two approaches: (a) preemptive priority discipline, where the less important task is preempted when the more important becomes ready, (b) processor sharing discipline, where the ready tasks run concurrently, interleaved in small time slices. We chose the latter because (1) the scheduling process is simpler (i.e., faster), (2) the runtime system support is simpler as no task ordering is required, and (3) frequent context switching overhead is low if one of the available light-weight thread packages is used (e.g., PThreads [5]).

For shared-memory systems it has been proven that an optimal preemptive task schedule is indeed shorter than non-preemptive schedules [2]. In the distributed case however, there is no such proof, nor we are aware of compile-time task scheduling algorithms specifically designed for distributed-memory preemptive environments.

In this paper we show that the existing scheduling algorithms for the non-preemptive case suffer considerable loss of performance when executed in a preemptive runtime

* This research is part of the Automap project granted by the Netherlands Computer Science Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO) under grant number SION-2519/612-33-005.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 272–276, 2000.
© Springer-Verlag Berlin Heidelberg 2000

environment based on a processor sharing discipline. A new scheduling algorithm for preemptive environments called the Preemptive Tasks Scheduling algorithm (PTS) is presented. PTS is focussed towards coarse-grain applications, as it aims to maintain an even processors load throughout the time rather than to reduce communication costs. Although PTS does not quite exploit the potential advantages of a preemptive scheme, it combines the low-cost of the algorithms for the non-preemptive case with a simpler runtime support, while the output performance is still at a level comparable to the non-preemptive schedules.

This paper is organized as follows: The next section describes the scheduling problem and introduces some definitions used in the paper. Section 3 presents the PTS algorithm, while Section 4 describes its performance. Section 5 concludes the paper.

2 Preliminaries

A parallel program can be modeled by a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of V nodes and $\mathcal{E}$ is a set of E edges. Nodes depict tasks, while edges represent inter-task communication. If two tasks are scheduled on the same processor, the communication cost between them is assumed to be zero. The length of a path is the sum of the computation and communication costs of the tasks and edges belonging to the path, respectively. The task's *bottom level* is the length of the longest path from the current task to any exit task. A task is said to be *ready* if all its parents have finished their execution. A task can start its execution only after all its messages have been received. The objective of the scheduling problem is to find a schedule of the tasks in $\mathcal{V}$ on the processors in $\mathcal{P}$ such that the parallel completion time (schedule length) is minimized.

3 The PTS Algorithm

Essentially, PTS schedules tasks in the order of their bottom levels on the least loaded processor (i.e., processor with the lowest number of tasks running on it). However, selecting the least loaded processor at a low cost is not a trivial problem. The processor load is given by the number of tasks simultaneously running on that processor. The time a task is running on its assigned processor is given not only by its size, but also by the processor load. Consequently, we need to simulate the preemptive execution of the tasks in order to compute the current processor load when scheduling a new task.

The PTS algorithm is formalized in Figure 1. Details about our simulation scheme and other implementation issues can be found in [7].

First, the tasks' priorities (bottom levels) are computed. Then, PST starts scheduling one ready task at a time. At each iteration, the ready task with the highest bottom level is selected. Using tasks' bottom levels ensures that the tasks are scheduled in the correct order with respect to dependencies.

Before scheduling the current task t, the task execution simulation is updated by stopping the tasks that finish before t's last message arrival time. As a consequence, the processor loads are also updated. Then, t is scheduled on the least loaded processor.

The PTS's complexity is $O(V(\log(V) + \log(P)) + E)$ as explained in [7].

```
PTS ()
BEGIN
    For all tasks compute bottom levels.
    WHILE NOT all tasks scheduled DO
        t ← Task with the highest bottom level.
        MAT ← t's last message arrival time.
        Stop the tasks finishing before MAT
            and update their successors.
        ST ← t's start time.
        p ← Least loaded processor.
        Start task t on p at ST.
    END WHILE
END
```

Fig. 1. The PTS algorithm

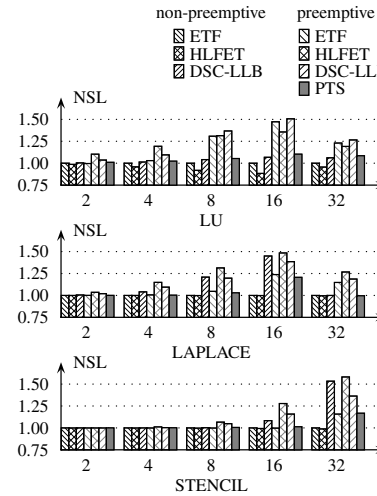


Fig. 2. Performance comparison

4 Performance Results

In this section we first investigate the loss of performance of three well-known non-preemptive scheduling algorithms (ETF [3], HLFET [1] and DSC-LLB [9, 8]) when simply applied to a preemptive environment. Next, we show the PST performance compared to the three above-mentioned algorithms in terms of both schedule lengths and execution times, but now run on a non-preemptive environment, for which they were originally designed.

We consider task graphs representing various types of parallel algorithms. The selected problems are *LU decomposition* ("LU"), *Laplace equation solver* ("Laplace") and a *stencil algorithm* ("Stencil"). For each of these problems, we adjusted the problem size to obtain task graphs of about 2000 nodes. As we consider the coarse-grain case, we generate task graphs with a communication to computation ratio (CCR) of 5. For each problem we generate 5 graphs with random execution times and communication delays (i.i.d. uniform distribution with unit coefficient of variance).

For performance comparison, we use the *normalized schedule length* (NSL), which is defined as the ratio between the schedule length of the given algorithm and the schedule length of a reference algorithm. As a reference algorithm we use ETF, applied to a non-preemptive environment. Schedule lengths are obtained by simulating the problems' execution in a homogeneous distributed system.

Scheduling Performance: As to be expected, in Figure 2 it can be seen that ETF, HLFET and DSC-LLB suffer a performance degradation when applied in a preemptive runtime environment. One can note that all ETF, HLFET and DSC-LLB algorithms yield schedules up to 47%, 53% and 41% longer in the preemptive case compared to the non-preemptive case. In contrast, PTS consistently yields better performance in the preemptive case. The figure also shows that PTS has schedule lengths comparable with

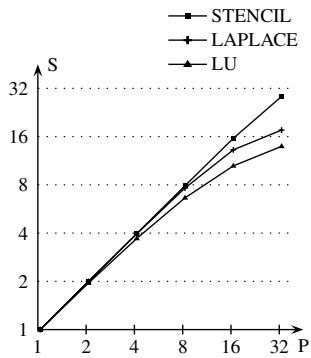


Fig. 3. PTS speedup

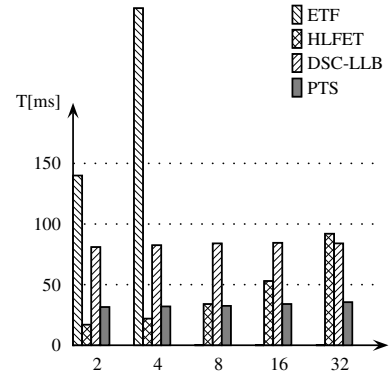


Fig. 4. Scheduling algorithm cost

those produced by ETF and HLFET in the non-preemptive case (for which they have been designed), and even improves DSC-LLB's schedule lengths with up to 31%.

Speedup: In Figure 3 we show the PTS speedup for the considered problems. For all the problems, PTS obtains significant speedup. For LU and Laplace, there are a large number of join operations. As a consequence, there is not much parallelism available and the speedup is lower. Stencil is more regular. Therefore more parallelism can be exploited and better speedup is obtained.

Running Times: In Fig. 4 the average running time of the considered algorithms is shown in CPU seconds as measured on a Pentium Pro/233MHz PC with 64Mb RAM. One can note that PTS's overall running time is the lowest, varying around 31 ms.

5 Conclusion

In this paper we investigate the potential advantages of compile-time task scheduling in a preemptive runtime system. A new scheduling algorithm, called Preemptive Task Scheduling algorithm (PTS) is presented, that is specifically designed for compile-time scheduling of coarse-grain problems in a preemptive runtime environment. PTS primarily focuses on obtaining a better processor utilization at a very low-complexity ($O(V(\log(V) + \log(P)) + E)$).

Experiments show that PTS performs comparable to ETF and HLFET, two top scheduling algorithms in the non-preemptive case, and outperforms DSC-LLB. Although, our results indicate that the potential advantages of using a preemptive scheme have not yet been exploited, PTS requires a simpler preemptive runtime management, as no task ordering is required. Further research will be performed to improve the scheduling algorithms in the preemptive case in order to outperform non-preemptive scheduling algorithms.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] E. G. Coffman Jr. *Operating Systems Theory*. Prentice Hall, 1973.

[3] J-J. Hwang, Y-C. Chow, F. D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. on Computing*, 18:244–257, 1989.

[4] Y-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proc. IPPS/SPDP*, 1998.

[5] F. Mueller. A library implementation of POSIX threads under UNIX. In *Proc. Winter*, 1993.

[6] A. Rădulescu and A. J. C. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proc. ACM ICS*, 1999.

[7] A. Rădulescu and A. J. C. van Gemund. Preemptive task scheduling for distributed systems. TR 1-68340-44(2000)04, Delft Univ. of Technology, 2000.

[8] A. Rădulescu, A. J. C. van Gemund, and H-X. Lin. LLB: A fast and effective scheduling algorithm for distributed-memory systems. In *Proc. IPPS/SPDP*, 1999.

[9] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, 1994.

Towards Optimal Load Balancing Topologies*

Thomas Decker, Burkhard Monien, and Robert Preis

Department of Mathematics and Computer Science, University of Paderborn, Germany
{decker,bm,robsy}@uni-paderborn.de

Abstract. Many load balancing algorithms balance the load according to a certain topology. Its choice can significantly influence the performance of the algorithm. We consider a two phase balancing model. The first phase calculates a balancing flow with respect to a topology by applying a diffusion scheme. The second phase migrates the load according to the balancing flow. The cost functions of the phases depend on various properties of the topology; for the first phase these are the maximum node degree and the number of eigenvalues of the network topology, for the second phase these are a small flow volume and a small diameter of the topology. We compare and propose various network topologies with respect to these properties. Experiments on a Cray T3E and on a cluster of PCs confirm our cost functions for both balancing phases.

1 Introduction

Load balancing algorithms are typically based on a fixed topology which defines the load balancing partners in the system. Only processors that are neighbors in the topology exchange both information and load items during the load balancing process.

There are three main aspects that influence the choice of a load balancing topology. Firstly, their fitness for the application. Topologies can reveal data-dependencies between tasks. This allows the load balancing algorithm to preserve the locality of the tasks. Secondly, their fitness for the hardware. Topologies can reveal the structure of the communication network of the parallel machine and thus reduce the total network load generated by the load balancing algorithm. Thirdly, their fitness for the load balancing algorithm. In order to speed up the load balancing phases of the application, a topology can be chosen that allows a fast convergence of the load balancing algorithm.

We focus on the third aspect. Absence of data dependencies and negligible influence of the interconnection topology are assumptions that are valid for a wide range of applications and architectures. Moreover, we consider a static load balancing problem. Thus, during the load balancing phase, no load is processed or generated.

In the case of load distributions that do not require global movements, load balancing algorithms that involve only nearest-neighbor communication are potentially superior to algorithms that involve global communication [15]. The conceptually simplest local iterative load balancing algorithm is the *First Order Diffusion* scheme (FOS) introduced by Cybenko [4]. All commonly used diffusion schemes generate the unique

* Supported by German Science Foundation (DFG) Project SFB-376, EU ESPRIT LTR Project 20244 (ALCOM-IT) and by EU TMR-grant ERB-FMGE-CT95-0052 (ICARUS 2).

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 277–287, 2000.
© Springer-Verlag Berlin Heidelberg 2000

l_2-minimal flow [6]. However, if diffusive schemes are used for direct load movement, they typically shift much more load items than necessary [6]. Therefore, the load balancing process is split into two phases. The first phase operates on a virtual load measure and, since this phase actually computes a network flow, we call it *flow computation phase*. In the *migration phase*, the real load items are distributed according to the computed flow.

Algorithms for the flow computation phase have been studied extensively. Many of them are local iterative schemes based on dimension exchange or diffusion [6, 7, 9, 12, 11, 18]. We consider the optimal diffusion scheme OPT [7] which computes the l_2-minimal flow. OPT is very simple and does only need $m - 1$ iterations where m is the number of distinct eigenvalues of the Laplacian matrix. As an extension, we introduce a *multiple diffusion scheme* (MD) which can be applied to topologies that are cartesian products of other graphs. MD applies the OPT scheme to each of the factor graphs one by one. Although the flow of MD is not necessarily l_2-minimal, the number of iterations can be decreased dramatically. Overall, we show that the time requirement of the flow computation phase depends on the maximum node degree and on the number of eigenvalues of the network.

During the migration phase we make use of the *Proportional Parallel Greedy*-heuristic (PPG) [6] that sends load preferably to that neighbor node that represents the largest sink. We show that both a small flow volume and a small diameter of the graph are important for this phase.

Overall, there is a demand for networks with small degrees, small numbers of distinct eigenvalues and small diameters for any value of node numbers. Furthermore, the networks should have the property that the applied balancing scheme can compute a flow with a small volume in the network. We discuss several graph classes such as cycles, hypercubes, cliques, tori or Cages. As we will see, the hypercubes have reasonably small values in all our measures. The hypercubes can be represented as many different cartesian products. By using the MD scheme with different numbers of dimension one can reduce the number of eigenvalues and establish a tradeoff between the time for calculating the balancing flow and the volume of the flow for the hypercube. There are special graphs for certain numbers of nodes such as the Petersen and the Hoffmann-Singleton [1] graphs with very low values for our measures, but they do not belong to a scalable graph class with similarly small values. Thus, our investigations are a first step to develop optimal load balancing topologies.

We integrated the algorithms into the load balancing library VDS [5]. We consider two platforms in order to cover the characteristics of both closely and loosely coupled systems. The Cray T3E captures the typical properties of massively parallel systems since its MPI implementation offers short message startup-times and small latency. Secondly, we consider a network of workstations (NOW) consisting of 96 Pentium II processors connected by Ethernet. The communication is based on PVM. The per-word transfer time of the NOW is about 15 times slower than for the Cray.

The following section defines the balancing flow and discusses the load balancing schemes OPT and MD. In Section 3 we develop a cost function for the flow calculation phase and recommend several topologies. The cost function of the migration phase together with several experiments on different load scenarios is discussed in Section 4.

2 Definitions and Background

Balancing Flow We represent the topology of the network by a connected, undirected graph $G = (V, E)$ with $|V| = n$ nodes and $|E| = N$ edges. Let $w_i \in \mathbb{R}$ be the *load* of node $v_i \in V$ with vector $w = (w_1, \ldots, w_n)$. Denote with $\tilde{w} := \frac{1}{n}\sum_{i=1}^{n} w_i$ the average load and vector $\overline{w} := (\tilde{w}, \ldots, \tilde{w})$. Let $\boldsymbol{E}$ be the same edge set as E, but with an arbitrary fixed direction to represent the direction of the flow. Let x_e represent a flow on $e \in \boldsymbol{E}$ with $x \in \mathbb{R}^N$. We call x a *balancing flow* for a load w, if for all $v_i \in V$ it is $w_i + \sum_{e=(v_j,v_i)\in\boldsymbol{E}} x_e - \sum_{e=(v_i,v_j)\in\boldsymbol{E}} x_e = \tilde{w}$. Consider the quality measures

- $l_1(x) = \|x\|_1 = \sum_{e\in\boldsymbol{E}} |x_e|$, the total costs of communication.
- $l_2(x) = \|x\|_2 = \sqrt{\sum_{e\in\boldsymbol{E}} x_e^2}$. This is minimized by common diffusion algorithms.
- $l_\infty(x) = \|x\|_\infty = \max_{e\in\boldsymbol{E}} |x_e|$, the max. communication between any two nodes.
- node-flow $f(x) = \max_{v_i\in V} f(v_i)$ with $f(v_i) = \sum_{e=\{v_i,v_j\}\in E} |x_e|$. This is the max. communication at any node (in- and out-going flow, see also [16]).

We define the *migration graph* $MG(x) = (V, \tilde{E})$ with directed edges by $(v_i, v_j) \in \tilde{E}$ iff $(v_i, v_j) \in \boldsymbol{E} \wedge x_{(v_i,v_j)} > 0$ or $(v_j, v_i) \in \boldsymbol{E} \wedge x_{(v_j,v_i)} < 0$. The *migration flow* $\tilde{x} \in \mathbb{R}^N$ on $\tilde{E}$ is defined by setting $\tilde{x}_{(v_i,v_j)} = x_{(v_i,v_j)}$ if $(v_i, v_j) \in \boldsymbol{E}$, and $\tilde{x}_{(v_i,v_j)} = -x_{(v_j,v_i)}$ otherwise. It is obvious that the migration graphs of l_1- and l_2-minimal flows are directed acyclic graphs (DAGs).

The balancing flow may also be defined in matrix notation. Let $Z \in \{-1, 0, 1\}^{n\times N}$ be the node-edge *incidence matrix* of G. Every column has two non-zero entries 1 and -1 for the two nodes incident to the edge. The signs define the direction of the edges according to $\boldsymbol{E}$. x is a balancing flow iff $Zx = w - \overline{w}$. Let $A \in \{0, 1\}^{n\times n}$ be the symmetric *adjacency matrix* of G. The *Laplacian* $L \in \mathbb{Z}^{n\times n}$ of G is defined as $L := C - A$, with $C \in \mathbb{N}^{n\times n}$ containing the degrees as diagonal entries.

Overview of Load Balancing Schemes In [12], the set of linear equations $Ly = w - \overline{w}$ is directly solved for y and the balancing flow is then derived by $x = Z^T y$. Executing this globally on one node requires a large amount of communication for gathering and broadcasting. The system $Zx = w - \overline{w}$ can also be calculated in parallel by using a standard conjugate gradient algorithm [12]. Besides, a matrix-vector multiplication (can be done locally in some iterations) requires three global summations of scalars.

A different way is to iteratively balance the load of a node with its neighbors locally until the whole network is globally balanced. There are two major approaches. In the *diffusion* schemes [4] each node sends and receives the current load of all of its neighbors simultaneously. All commonly used diffusion schemes calculate the unique l_2-minimal flow [6]. Thus, the migration graph of a diffusion flow is a DAG. In the *dimension exchange* (DE) schemes [4, 18] each node balances its load with one neighbor after another. The flow of a DE scheme is not necessarily l_2-minimal.

Some further models can be used for networks that are a cartesian product of graphs such as tori or hypercubes. Here, $G = G_1 \times G_2$ with node set $V(G) = V(G_1) \times V(G_2)$ and edge set $E = \{((u_i, u_j), (v_i, v_j)); u_i = v_i \wedge (u_j, v_j) \in E(G_2) \text{ or } u_j = v_j \wedge (u_i, v_i) \in E(G_1)\}$. Cartesian products of lines and paths were previously discussed in [18]. The *Alternating Direction Iterative* (ADI) scheme [7] is a combination of diffusion

and dimension exchange for cartesian products. It is a common method of solving linear systems [17] that is applied to the load balancing problem. In each iteration a node first communicates with its neighbors according to G_1 and then with its neighbors according to G_2. Unfortunately, the resulting flow may not be a DAG, leading to fairly high flow values. In the *multi diffusion* (MD) scheme a node exchanges the load iteratively with its neighbors in dimension 1 until the subgraph G_1 to which it belongs is balanced. Then, it balances along dimension 2 until the whole network is balanced. For dimension 1 there are $|V(G_2)|$ and for dimension 2 there are $|V(G_1)|$ independent load balancing tasks. We use a standard diffusion scheme for G_1 and G_2 for these tasks.

Theorem 1. *Let graph G be a cartesian product $G = G_1 \times G_2$ and let x be a balancing flow of G calculated with the MD scheme using a sub-scheme which calculates a migration DAG for both directions. Then, the migration graph $MG(x)$ is a DAG, too.*

Proof. Assume it has a cycle. At least one edge of the cycle has to be in dimension 2, because dimension 1 is acyclic, due to the assumption. After having finished the balancing in dimension 1, all $|V(G_2)|$ subgraphs that are isomorphic to G_1 are balanced within themselves. Thus, the load balancing flow in dimension 2 has to have a cycle. This is false, because all $|V(G_1)|$ independent balancing tasks in dimension 2 are acyclic and, because of the same load distribution, all of them have the same flow direction. □

The MD scheme can easily be generalized for d-dimensional cartesian products $G_1 \times \ldots \times G_d$. Theorem 1 can be extended such that MD guarantees a DAG for any d.

Diffusive Load Balancing Schemes These schemes perform iterations with communication between adjacent nodes. In the first-order-scheme (FOS) [4, 9] node $v_i \in V$ performs $w_i^k = w_i^{k-1} - \sum_{\{v_i,v_j\}\in E} \alpha(w_i^{k-1} - w_j^{k-1})$ with flow $x^k_{e=\{v_i,v_j\}} = x_e^{k-1} + \alpha(w_i^{k-1} - w_j^{k-1})$. Here, w_i^k is the load of node i after k iterations, and x_e^k is the total load sent via edge e until iteration k. For an appropriate choice of α, FOS converges to the average load $\overline{w}$ [4]. The resulting flow is l_2-minimal [6].

FOS can be transcribed as $w^k = Mw^{k-1}$ with $M = I - \alpha L \in \mathbb{R}^{n\times n}$. Let $0 = \lambda_1 < \ldots < \lambda_m$ be the m distinct eigenvalues of the Laplacian L. It is known that $\lambda_1 = 0$ is simple (for a connected graph G) with eigenvector $(1, \ldots, 1)$ [1]. Furthermore, the minimum number of iterations for FOS can be obtained by $\alpha = \alpha_{opt} = \frac{2}{\lambda_2+\lambda_m}$. The main disadvantage of FOS is its slow convergence, even when α_{opt} is used. Furthermore, FOS never achieves the exactly balanced load of $\overline{w}$. FOS has to be terminated when the imbalance is small enough. This process requires a global termination detection.

A numerically stable optimal scheme OPS was introduced in [6]. OPS only needs $m-1$ iterations and balances the load vector to exactly $\overline{w}$. It is also a diffusion scheme and calculates the l_2-minimal flow. Another optimal scheme OPT [7] with the same convergence properties was derived from OPS. Although OPT might trap in a numerical unstable situation, it was shown of how to avoid them. The difference between OPT and FOS iterations is the fact that the parameter α varies for each iteration. For iteration k we choose $\alpha = \alpha_k = \frac{1}{\lambda_k}$ with λ_k, $2 \le k \le m$ being the distinct non-zero eigenvalues of L. For each iteration a different eigenvalue of L is used. The order of the eigenvalues is arbitrary. We choose an order that achieves numerical stability [7]. The main disadvantage of OPT is the fact that all eigenvalues of the graph are required. Although the

Table 1. Graphs with $m - 1 = D$.

Graph	$\|V\|$	degrees	Spectrum of L_G	$m - 1 = D$
Path(n)	n	$1, 2$	$2 - 2 \cdot \cos(\frac{\pi}{n} j) \quad j = 0, \ldots, n-1$	$n - 1$
Cycle(n)	n	2	$2 - 2\cos\left(\frac{2\pi}{n} j\right) \quad j = 0, \ldots, n-1$	$\lfloor \frac{n}{2} \rfloor$
Star(n)	n	$1, n-1$	$0, 1, n$	2
complete k-partite(n)	n	$n - \frac{n}{k}$	$0, deg^{[n-k]}, n^{[k-1]}$	2
Clique(n)	n	$n - 1$	$0, n^{[n-1]}$	1
Hyp(d)	2^d	d	$2j \quad j = 0, \ldots, d$	d
Lattice(k, d)	k^d	$d(k-1)$	$jk \quad j = 0, \ldots, d$	d
Petersen/$deg3D2$/MCage(3,5)	10	3	0,2,5	2
Hoff.-Sing./$deg7D2$/MCage(7,5)	50	7	0,5,10	2
MCage(d, 6), $d - 1$ prime-power	$\frac{2(d-1)^3-2}{d-2}$	d	$0, d \pm \sqrt{d-1}, 2d$	3
MCage(d, 8), $d - 1$ prime-power	$\frac{2(d-1)^4-2}{d-2}$	d	$0, d \pm \sqrt{2(d-1)}, d, 2d$	4
MCage(d, 12), $d - 1$ prime-power	$\frac{2(d-1)^6-2}{d-2}$	d	$0, d \pm \sqrt{3(d-1)}, d \pm \sqrt{d-1}, d, 2d$	6

calculation of the eigenvalues can be time-consuming for large graphs, they can easily be computed for small graphs. Besides, they do only have to be calculated once for each graph and can be applied for any load situation. Furthermore, they are known for many classes of graphs that often occur as processor networks (see e. g. [3, 12]). Another optimal scheme is presented in [11].

As a byproduct, the optimal schemes lead to $D \leq m - 1$ with D being the diameter of the graph. This fact has already been proven in a different way in [1, 3].

3 Flow Calculation

In this section, we discuss the first step of the balancing process, i. e. the calculation of the balancing flow. For cartesian products of graphs we also use the scheme MD in combination with OPT for each dimension. In each iteration of OPT a node has to send the information about its current load to all neighbors and it has to receive their load. Thus, its number of send/receive-operations per iteration is equal to its degree. We mainly discuss graphs $G = (V, E)$ with regular degree $deg(G)$. As discussed above, the number of iterations of OPT is $m - 1$ with m being the number of non-zero distinct eigenvalues of the Laplacian of G. Thus, every node executes a total of $(m-1) \cdot deg(G)$ send- or receive-operations. The MD scheme for the cartesian product $G = G_1 \times \ldots \times G_d$ of graphs G_i involves $(m_i - 1) \cdot deg(G_i)$ operations in dimension i for every node. The total number of send- or receive-operations for each node is $M(G, d) = \sum_{i=1}^{d} (m_i - 1) \cdot deg(G_i)$. Thus, networks with small deg and m are desirable. As stated in section 2, it holds $m - 1 \geq D$ with D being the diameter of the graph. Table 1 lists some graphs with $m - 1 = D$ (see also [1, 3, 12]).

Paths, cycles, stars, complete k-partite graphs and cliques either have a high maximum degree or a high number of distinct eigenvalues. Hypercubes have logarithmic values for both measures. The Lattice graphs are hypercubes over an alphabet of order k. Unfortunately, there are only hypercubes and Lattice graphs for some node numbers.

A well known problem is the construction of (deg, D)-graphs which are the graphs with the largest number of nodes for a given degree deg and diameter D [2]. A (deg, D)-graph has at most $\frac{deg(deg-1)^D-2}{deg-2}$ nodes (*Moore-Bound*). This bound is attained for

Graph G	dim. d	$deg(G_1)$	$m_1 - 1$	$M(G,d)$
Cycle(64)	1	2	32	64
Clique(64)	1	63	1	63
Torus(8,8)	1	4	12	48
Cycle(8)^2 (MD)	2	2	4	16
Hyp(6)	1	6	6	36
Hyp(2)^3 (MD)	3	2	2	12
Hyp(1)^6 (MD)	6	1	1	6
Lattice(4,3)	1	9	3	27
Lattice(4,3) (MD)	3	3	3	9
Butterfly(4)	1	4	10	40
DeBruijn(6)	1	4	17	68
Knödel(64)	1	6	33	198
Cage(6,6)	1	6	3	18
Knödel(62)	1	5	7	35

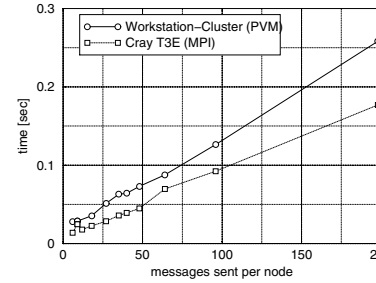


Fig. 1. Left: characteristics of graphs. The cartesian products have the identical subgraph G_1 with $M(G,d) = d(m_1 - 1)deg(G_1)$. Right: flow computation costs with respect to the number of messages sent per node. The times measured for the Cray T3E are scaled by a factor of 10.

cliques. For $d \geq 3$ and $D \geq 2$ it is only attained if $D = 2$ and $deg = 3$ (Petersen graph), $deg = 7$ (Hoffmann-Singleton graph, [1]) and (perhaps) $deg = 57$. There is no scalable construction for (deg, D)-graphs. Our experiments revealed that among the largest known (deg, D)-graphs only the stated graphs have a fairly small value of m. Related graphs are (deg, g)-*Cages* which are the smallest graphs with degree deg and shortest cycle (*girth*) g [1]. A Cage has at least $\frac{deg(deg-1)^{(g-1)/2}-2}{deg-2}$ nodes for an odd g and at least $\frac{2(deg-1)^{g/2}-2}{deg-2}$ nodes for an even g. A Cage with a size equal to this bound is called *minimum Cage* MCage(deg,g). There are only few graphs of this kind, such as the mentioned (deg, D)-graphs that achieve the Moore bound, cycles and complete bipartite graphs. Further MCages(deg,g) do only exist for $g = 6, 8, 12$ and $deg - 1$ prime power. For MCages it holds $m - 1 = D = \lfloor \frac{g}{2} \rfloor$. The sizes of (deg, g)-Cages are only known for a limited number of values of deg and g [14]. Our experiments revealed that among the known Cages only the MCages have fairly small values of m.

The Knödel graphs [13] can be constructed for any number of nodes. Their degree is $\lfloor \log_2 n \rfloor$ and their diameter is at most $\lceil \log_2 n \rceil$. For $n = 2^i$ with some $i \in \mathbf{N}$, their diameter is $\lceil \frac{\log_2 n+2}{2} \rceil$ [8]. This is an improvement over the hypercube. It has been shown that Knödel graphs of size $2^i - 2$ with $i > 1$ are edge-symmetric [10]. Our experiments show that only these Knödel graphs have a fairly low number of eigenvalues.

Fig. 1(left) presents a comparison of several graphs with 64 nodes (except for Cage(6,6) and Knödel(62) with 62 nodes). We also considered the 8×8 torus, the butterfly graph of dimension 4 and the DeBruijn graph of dimension 6. Several graphs can be expressed as a cartesian product such as Hyp(6) = Hyp(2)^3 = Hyp(1)^6. Here, MD can be performed in 1, 2 or 3 dimensions with different values $M(G,d)$. $M(G,d)$ decreases with increasing d, but the amount of flow increases (Sec. 4). For $d = 1$ the value $M(G,d)$ is higher, but the diffusion scheme calculates the l_2-minimal flow.

Fig. 1(right) presents the relation between $M(G,d)$ and the execution times of the flow computation for various graphs. The linear progress shows that $M(G,d)$ is a prac-

ticable cost function and, by applying linear regression, we obtain $T_{flow}(Cluster) \approx 1.2\text{ms} \cdot M(G,d) + 17\text{ms}$ and $T_{flow}(CrayT3E) \approx 0.084\text{ms} \cdot M(G,d) + 0.98\text{ms}$.

4 Flow Migration

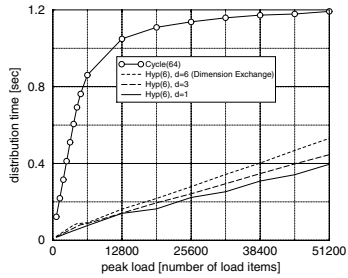
Once the flow is computed, the nodes start with the migration phase. Let $v \in V$ be a node with neighbors $v_1, \ldots, v_k$ in the migration graph. The node has to send a load of $\tilde{x}_{(v,v_i)}$ to neighbor v_i. Since a node v might need to send this load in several steps, it keeps track of the remaining number of load items out_{v,v_i} which have to be sent to v_i (out_{v,v_i} is initialized with $\tilde{x}_{(v,v_i)}$ rounded to the closest integer value). Each time node v sends load items to v_i, it updates out_{v,v_i}. This distribution algorithm is executed once the flow is computed and again whenever new load items arrive. The load migration phase terminates as soon as $out_{v,w} = 0$ for all $(v,\, w) \in \tilde{E}$. This condition is met after a finite number of steps (since the flows generated by MD are acyclic).

We introduce logical communication-rounds in order to determine the duration of the load migration phase. Let $r(v)$ denote the round-number of a node $v \in V$. Initially, $r(v) = 0$ for all $v \in V$. Each message is tagged with the round-number of the sending node plus one. Whenever a node v receives a message, its round-number $r(v)$ is set to the maximum of $r(v)$, and the round-number of the incoming message. We define the maximal round number $r := \max_{v \in V} r(v)$. Denote with $out_v := \sum_{(v,w) \in \tilde{E}} out_{v,w}$ the *out-going* flow of v. If out_v is not larger than the initial load w_v, the total out-going flow can be saturated in one step. A *conflict* occurs whenever out_v is larger than the current load of v. In this case, we have to decide on how much load to send to each of the neighbors. It is known that r is bounded by $\mathrm{O}(\sqrt{n})$ for all local greedy algorithms which always migrate all available load items [6]. The PPG-heuristic belongs to this class of algorithms. PPG moves the portion $out_{v,v_i}/out_v$ of the current load to node v_i. Consequently, the load is preferably moved in the direction of the largest sink.

We assume that the time needed to send or receive a message of size s is given by $t_o + s\, t_w$. Parameter t_o models a constant communication overhead (such as the startup time of the communication network). Parameter t_w models the per-word transfer time. We can bound the duration T_m of the migration phase as $T_m \leq (r \deg(G))\, t_o + f(x)\, t_w$. Both values r and $f(x)$ depend on the balancing flow which again depends on the topology and the balancing scheme.

In the following, we use two load scenarios and illustrate the cost function for several topologies. In the *peak*-scenario, we set w_1 to a positive value and $w_i = 0$, $2 \leq i \leq n$. Thus, the peak-scenario models applications which incorporate only few nodes with the majority of the load. In order to model applications that have a more balanced generation pattern, we use the *random*-scenario and draw the entries of w from a uniform random distribution.

The peak-scenario. Fig. 2(left) presents the timings for the migration phase that were obtained on the Cray T3E. Up to 51200 load elements were generated on node 0 (each load object consists of 150 bytes). The diagram shows five topologies with 64 nodes each, namely the cycle, three schemes for the hypercube, and the clique. In all cases, r is equal to the diameter of the network. Interestingly, this is not necessarily the case [6]. Moreover, for a fixed initial load, the node flow $f(x)$ is always the same. The



G	r	$l_2(x)$	migration time	
			Cray	NOW
Cycle(64)	32	35638	1.19	6.9
Hyp(6)	6	22755	0.40	2.1
Hyp(2)^3	6	32790	0.47	2.1
Hyp(1)^6	6	35919	0.53	2.1
Clique(64)	1	6350	0.38	2.0

Fig. 2. Duration of migration phase for various topologies and peak load. The diagram on the left displays the dependency of the duration of the migration phase on the extent of the peak load. The experiments were conducted on a Cray T3E. The table on the right relates the migration time to the diameter of the topologies and the l_2-norm of the flow. The times are given in seconds.

long duration of the migration time in case of the cycle topology is due to the large diameter. It can be reduced significantly, if we choose networks with smaller diameter.

The three timings for the hypercube refer to the different schemes, namely diffusion, multi diffusion (MD) with respect to Hyp$(2)^3$ and Hyp$(1)^6$ (dimension exchange). They differ in the way they distribute the load along the edges. The number of edges that take part in the process depends on the number of dimensions of the MD scheme. For the original diffusion, all edges are used. If we apply the MD scheme, only some of the edges are used. For example, the dimension exchange method does only use 63 edges which is about one third of all edges. The unbalanced distribution of the communication load generates a critical path that is responsible for the duration of the migration phase.

The $l2$-norm of the flow is a measure for the balance of the communication load. Fig. 2 (right) lists the diameter, the $l2$-norm of the flow, and the migration time for an initial load of 51200 items. It reveals that T_m is mainly influenced by the diameter.

The random-scenario. The distance between high- and low-loaded processors involved by the random-scenario is much smaller than the distance involved by the peak-scenario. Thus, the number of migration rounds is typically much smaller than the diameter. Fig. 3 (left) lists the average number $\overline{r}$ of migration rounds that are needed for various topologies. Except for the clique and the cycle, about two migration rounds suffice to balance the load. Thus, in contrast to the peak-scenario, r is an unimportant parameter. The dominating parameter is the amount of data the nodes have to communicate, i. e. the maximum node flow $f(x)$.

Fig. 3 (right) presents a strong correlation between the node flow and the migration time. We cannot expect a perfect correlation, since the node flow does only represent the second term of the cost function T_m. The first term depends on r and the degree of the network. The node flow is too pessimistic for the cycle and too optimistic for the clique. One reason for these deviations are the extreme degrees of these graphs.

The network flows listed in Fig. 3 (left) reveal that the node-flow depends on the topology and on the balancing scheme. The MD schemes produce a larger node flow than the diffusion scheme because MD balances the dimensions of the topology one

G	$\overline{r}$	$f(x)$	$l_2(x)$	T_m Cray [s]	T_m NOW [s]	$T_m/f(x)$ Cray [ms]	$T_m/f(x)$ NOW [ms]	$T_m/l_2(x)$ Cray [μs]	$T_m/l_2(x)$ NOW [ms]
Cycle(64)	**3.3**	**4067**	**7591**	**0.076**	**0.54**	**0.018**	**0.13**	**9.6**	**0.07**
Clique(64)	**1.0**	**875**	**478**	**0.026**	**0.60**	**0.030**	**0.69**	**53.6**	**1.27**
Torus(8,8)	1.8	1195	2338	0.024	0.51	0.020	0.43	10.4	0.22
Cycle(8)^2, MD	2.2	1641	3239	0.029	0.61	0.019	0.37	9.6	0.19
Clique(4)^3	1.3	929	1345	0.022	0.44	0.024	0.47	16.8	0.32
Clique(4)^3, MD	2.1	1950	1919	0.027	0.60	0.022	0.48	14.4	0.32
Hyp(6)	2.1	970	1679	0.022	0.49	0.023	0.50	13.6	0.29
Hyp(2)^3, MD	2.1	1282	2481	0.025	0.51	0.019	0.40	9.6	0.20
Hyp(1)^6, MD	1.3	1355	2733	0.028	0.64	0.020	0.47	10.4	0.23
Butterfly(4)	2.0	1214	2288	0.024	0.50	0.020	0.41	10.4	0.22
DeBruijn(6)	1.7	1219	2314	0.026	0.54	0.022	0.44	11.2	0.23
Gossip (64)	1.4	991	1676	0.021	0.52	0.022	0.53	12.8	0.31
Gossip (62)	1.4	998	1850	0.022	0.45	0.022	0.45	12.0	0.24
Cage(6, 6)	1.4	1063	1624	0.023	0.49	0.022	0.46	14.4	0.30

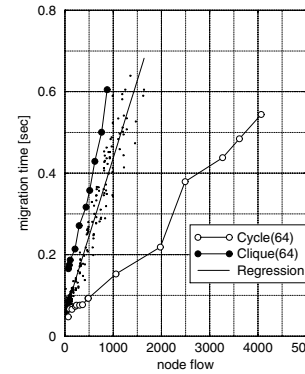


Fig. 3. Left: properties of the migration phase in case of the random load scenario. All entries are average values of 10 experiments with an average load of 800 load items. The diagram on the right correlates the node flow to the migration time measured on the PC-Cluster. The diagram is based on 300 experiments with various topologies including those listed in table on the left. Each point represents the average node flow and the corresponding average migration time of 10 experiments with the same initial total load.

after another. For example, consider the Cycle(8)^2 topology. In the first iteration, MD balances eight cycles in parallel. For a random load distribution, the system is balanced quite well after this first step. Experiments have shown that only about 20% of the total flow corresponds to the edges of the second dimension. This property has two consequences. Firstly, it leads to an unbalanced distribution of the communication load across the edges. We have already observed this effect in the scope of the peak scenario. Secondly, close migration partners for high loaded nodes may not be addressed in the first place since they are in a different dimension. This leads to unnecessary migrations and a high node flow. The diffusion scheme avoids these unnecessary migrations. Thus, schemes which use all edges in each step also generate small node flows (cf. the l_2-norm of the flows shown in Fig. 3(left)). Small values of $l_2(x)$ always imply small values of $f(x)$. We observe a strong correlation between these two measures. All experiments revealed that the quotient $f(x)/l_2(x)$ was always between 0.18 and 0.29. Its average value was 0.21. Thus, the diffusion schemes seem to generate flows that are close to optimal with respect to their node flow. In the case of the clique, the diffusion flow is in fact optimal with respect to the node flow.

Theorem 2. *The unique l_2-minimal balancing flow of the clique with n nodes and load vector $w \in I\!R^n$ has the node-flow $f(x) = \triangle w = \max_{1 \le i \le n} |w_i - \tilde{w}|$.*

Proof. The clique has the eigenvalues 0 and n. Thus, OPT calculates the l_2-minimal flow in one iteration and the flow over edge $\{u, v\}$ is $\frac{|w_u - w_v|}{n}$. The node-flow of a node v is $f(v) = \sum_{u \in V} \frac{|w_v - w_u|}{n}$. If $|\{u \in V; w_u \le w_v\}| < |\{u \in V; w_u > w_v\}|$, then $\sum_{u \in V} |w_v - w_u| \le \sum_{u \in V} |w_{min} - w_u|$, else $\sum_{u \in V} |w_v - w_u| \le \sum_{u \in V} |w_{max} - w_u|$, where w_{min} and w_{max} denote the minimum and maximum load of all nodes.

Thus, $f(v) = \sum_{u \in V} \frac{|w_v - w_u|}{n} \leq \max\{\sum_{u \in V} \frac{|w_{min} - w_u|}{n}, \sum_{u \in V} \frac{|w_{max} - w_u|}{n}\} \leq \triangle w$. Obviously, at least one node v has $f(v) \geq \triangle w$ which completes the proof. □

Thus, for a fixed number n of nodes and a load vector w, the l_2-minimal balancing flow of the clique has the minimal node-flow of any balancing flow on any topology.

Unfortunately, as we have seen before, the measure node-flow is too optimistic for the clique, due to the large degree. Fig. 3 shows that the migration time of the clique is larger than that of the diffusion scheme for the hypercube, although the node flow of the clique is smaller. Nevertheless, a small node flow is the main condition for a short migration phase; the l_2-optimal flow of the diffusion scheme implies a small node flow.

5 Conclusion

The choice of the topology and the load balancing scheme is closely related to the time requirement of the balancing process. For the flow-computation phase, a small *node degree* and a small *number of eigenvalues* of the network reduce the time requirement of the optimal scheme OPT. Besides, if the network can be represented as a cartesian product of several graphs, the scheme MD can be applied, using the much fewer eigenvalues of the factor graphs. The time-requirement of the migration phase depends on the load situation. In the peak-scenario, a small diameter of the topology is desirable. In the random scenario the migration phase is much faster and is dominated by the *node-flow*. Since the flows calculated by the diffusion schemes have low node-flow values, the diffusion schemes are particularly well suited.

Networks which simultaneously minimize the degree, the number of eigenvalues, the diameter, and the node-flow are desired. We proposed several graph classes, each of which minimizing some of these measures. Thus, our investigations in this paper are a first step to establish a set of topologies with small cost values.

References

[1] N. Biggs. *Algebraic Graph Theory, Second Edition*. Cambridge University Press, 1974/1993.
[2] F. Comellas. (degree,diameter)-graphs. http://www-mat.upc.es/grup_de_grafs/table_g.html.
[3] D.M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs*. Joh. Ambrosius Barth, 1995.
[4] G. Cybenko. Load balancing for distributed memory multiprocessors. *J. of Parallel and Distributed Computing*, 7:279–301, 1989.
[5] T. Decker. Virtual Data Space – Load balancing for irregular applications. *Parallel Computing*, 2000. To appear.
[6] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.
[7] R. Elsässer, A. Frommer, B. Monien, and R. Preis. Optimal and alternating-direction load-balancing schemes. In *EuroPar'99*, LNCS 1685, pages 280–290, 1999.
[8] G. Fertin, A. Raspaud, H. Schröder, O. Sykora, and I. Vrto. Diamater of Knödel graph. In *Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, 2000. to appear.
[9] B. Ghosh, S. Muthukrishnan, and M.H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *SPAA*, pages 72–81, 1996.

[10] M. C. Heydemann, N. Marlin, and S. Perennes. Cayley graphs with complete rotations. Technical Report 1155, L.R.I. Orsay, 1997.
[11] Y.F. Hu and R.J. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.
[12] Y.F. Hu, R.J. Blake, and D.R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Prac. and Exp.*, 10(6):467–483, 1998.
[13] W. Knödel. New gossips and telephones. *Discrete Mathematics*, 13:95, 1975.
[14] G. Royle. Cages of higher valency. http://www.cs.uwa.edu.au/∼gordon/cages/allcages.html.
[15] P. Sanders. Analysis of nearest neighbor load balancing algorithms for random loads. *Parallel Computing*, 25:1013–1033, 1999.
[16] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. of Parallel and Distributed Computing*, 47(2):109–124, 1997.
[17] R.S. Varga. *Matrix Iterative Analysis*. Prentice-Hall, 1962.
[18] C. Xu and F.C.M. Lau. *Load Balancing in Parallel Computers*. Kluwer, 1997.

Scheduling Trees with Large Communication Delays on Two Identical Processors

Foto Afrati[1], Evripidis Bampis[2], Lucian Finta[3], and Ioannis Milis[4]

[1] National Technical University of Athens, Division of Computer Science, Heroon Polytechniou 9, 15773 Athens, Greece
[2] LaMI, Université d'Evry, Boulevard des Coquibus, 91025 Evry Cedex, France
[3] LIPN, Université Paris 13, Avenue Jean-Baptiste Clément, 93430 Villetaneuse Cedex, France
[4] Athens University of Economics and Business, Department of Informatics Patission 76, 10434 Athens, Greece

Abstract. We consider the problem of scheduling trees on two identical processors in order to minimize the *makespan*. We assume that tasks have unit execution times, and arcs are associated with large identical communication delays. We prove that the problem is NP-hard in the strong sense even when restricted to the class of *binary trees*, and we provide a polynomial-time algorithm for *complete* binary trees.

1 Introduction

Two-processor scheduling is one of the most known problems in scheduling theory. It is a particular case of the famous m-processor scheduling problem where a graph of unit execution time (UET) tasks has to be scheduled on m identical processors in order to minimize the *makespan*.

If no communication delays occur between tasks in precedence relation, the two-processor scheduling problem is polynomial [4,6]. On the contrary, the complexity of the three-processor scheduling problem remains an outstanding open question [8].

This picture changes when we consider the two-processor scheduling problem with unit interprocessor communication delays. This variant of the problem is extensively studied. However, its complexity for arbitrary task graphs remains unknown and polynomial time optimal algorithms have been shown for several classes of task graphs, especially trees [12,14], interval orders [1] and series-parallel graphs (SP1) [5].

Although a large amount of work is concentrated on the unit communication delays case, no results are known for the two-processor scheduling problem with large communication delays. The only known results on scheduling in the presence of large communication delays concern the case where a sufficiently large number of processors is available [2,3,7,11,10,13].

In this paper we deal with the two-processor scheduling problem in the presence of large identical communication delays. Formally, we are given a set

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 288–295, 2000.
© Springer-Verlag Berlin Heidelberg 2000

$P = \{p_1, p_2\}$ of two identical processors and a set $V = \{1, 2, ..., n\}$ of partially ordered tasks represented by a directed acyclic graph (dag) $\mathcal{G} = (V, E)$, referred as *task graph*. Tasks have unit execution times (UET), denoted by $\mathsf{p}_j = 1$, and their execution is subject to precedence constraints and communication delays; whenever two communicating tasks i, j, with $(i, j) \in E$, are scheduled on different processors an identical communication delay $c_{ij} = c(n)$ is introduced. By C_{max} we denote the makespan (length) of a schedule, that is the last time unit some task is executed on any processor. According to the three-field notation scheme for scheduling problems, introduced in [9] and extended in [15], our problem is denoted as

$$P2 \mid trees,\ \mathsf{p}_j = 1,\ c_{ij} = c(n) \mid C_{\max}.$$

We prove that the problem is NP-hard even for binary trees, and we present a polynomial algorithm for complete binary trees. Our results show that the complexity behavior of the two-processor scheduling problem with large communication delays is analogous to the case where a sufficiently large number of processors is available.

2 NP-Hardness Result

To prove that $P2 \mid binary\ trees,\ \mathsf{p}_j = 1,\ c_{ij} = c \mid C_{max}$ is NP-hard, we give a reduction from the following special case of the well known NP-complete problem EXACT-COVER BY 3-SETS (X3C), that we call X3C1 [8]:

INSTANCE: A set $U = \{h_1, h_2, \ldots, h_{3m}\}$ and a family $F = \{S_1, S_2, \ldots, S_k\}$ of subsets of U such that $S_t = \{h_i, h_j, h_v\}$, $1 \leq t \leq k$, and every element of U belongs to no more than three elements of F.

QUESTION: Are there m elements of F whose union is exactly U?

For every instance of X3C1, we construct an instance of $P2 \mid binary\ trees$, $\mathsf{p}_j = 1,\ c_{ij} = c \mid C_{max}$ in the following way:

We choose constant integers G, G', a such that $G >> a >> G' >> m^3$.

For every element $S_t = \{h_i, h_j, h_v\}$ of F (w.l.o.g. we assume $i > j > v$), we construct a subtree T_t as shown in Figure 1, where $H_i = a(m^3 + i)$, $i = 1, 2, \ldots, 3m + 1$ (the lengths of its chains are depicted in the figure). Let us denote by $\mathcal{B}$ the total number of tasks in all trees T_t, $t = 1, \ldots, k$, i.e.

$$\mathcal{B} = \sum_{t=1}^{k} |T_t| = kG + kG' + \sum_{\forall S_t = \{h_i, h_j, h_v\} \in S} (H_i + H_j + H_v + H_{i+1} + H_{j+1} + H_{v+1}).$$

Given the subtrees T_t, $t = 1, \ldots, k$, we construct the tree $\mathcal{T}$ as shown in Figure 2 and we consider the following scheduling problem:

Can we schedule $\mathcal{T}$ with communication delay

$$c = \mathcal{B} + G - 2(\textstyle\sum_{i=1}^{3m} H_i + mG') - H_{3m+1},$$

on two identical processors p_1 and p_2 within deadline

$$D_m = k + 1 + \mathcal{B} - (\textstyle\sum_{i=1}^{3m} H_i + mG') + G?$$

We prove first that if X3C1 has a positive answer, then there exists a feasible schedule $\mathcal{S}$ such that $C_{\max}(\mathcal{S}) \leq D_m$. Indeed, let, w.l.o.g., $F^* = \{S_1, S_2, \ldots, S_m\}$

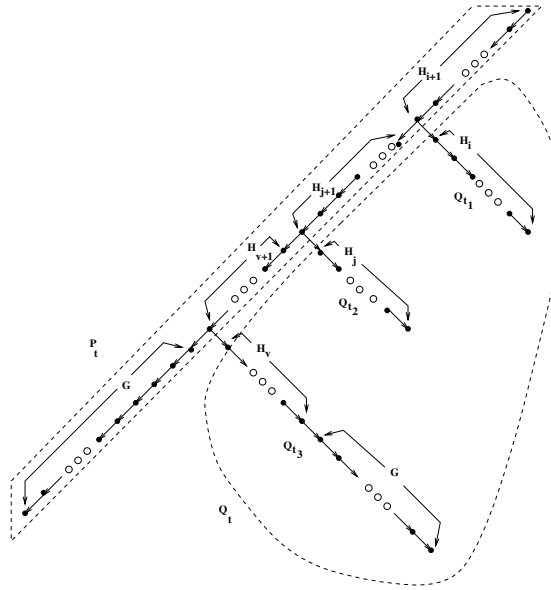


Fig. 1. *The subtree T_t corresponding to the element $S_t = \{h_i, h_j, h_v\}$ of F*

be a solution of X3C1, i.e. $|F^*| = m$, $F^* \subset F$ and $\bigcup_{i=1}^{m} S_i = U$. Let us also denote by $T_1, \ldots, T_m$ the subtrees of $\mathcal{T}$ corresponding to the elements of F^*. Then, a valid schedule is the following:

Processor p_1, starts at time 0 by executing the root of the tree. Then it executes the first k tasks of the path P_0 of $\mathcal{T}$ and then the chains $H_{i+1}, H_{j+1}, H_{v+1}$ of P_t, $1 \leq t \leq m$, in decreasing order of their lengths i.e. $H_{3m+1}, H_{3m}, \ldots, H_2$. These $3m$ chains can be executed in that way since by the construction of $\mathcal{T}$, after the execution of some chain, say H_{i+1} of some P_t, $1 \leq t \leq m$, the chain H_i of some $P_{t'}$, $1 \leq t' \leq m$, is always available. Processor p_1 finishes by executing the remaining G tasks of every path P_t, $t = 1, \ldots, m$, the tasks of the remaining subtrees T_t, of $\mathcal{T}$, $t = m+1, \ldots, k$, and the last G tasks of P_0. Processor p_2 starts executing at time $c + k + 1$ the H_{3m+1} tasks of Q_0 and then the tasks of the branches $Q_{t_1}, Q_{t_2}, Q_{t_3}$ of T_t, $1 \leq t \leq m$, in decreasing order of their lengths i.e. $H_{3m}, H_{3m-1}, \ldots, H_1$. A chain of Q_t of length H_l is always just in time to be executed on p_2, i.e. the first task of this chain is ready to be executed exactly at the end of the execution of the last task of the chain of $Q_{t'}$ of length H_{l+1} on the same processor. Notice that the last task of the chain of length H_{l+1} of P_t has been completed exactly c time units before on p_1. Finally, it executes the remaining G' tasks at the end of the Q_{t_3}'s branches of every T_t, $1 \leq t \leq m$ (the order is not important). Figure 3 illustrates the Gantt-chart of such a schedule.

Let us now show that if $\mathcal{T}$ can be scheduled within time D_m, then X3C1 has a solution F^*. In the following, we consider w.l.o.g. that the root of $\mathcal{T}$ is

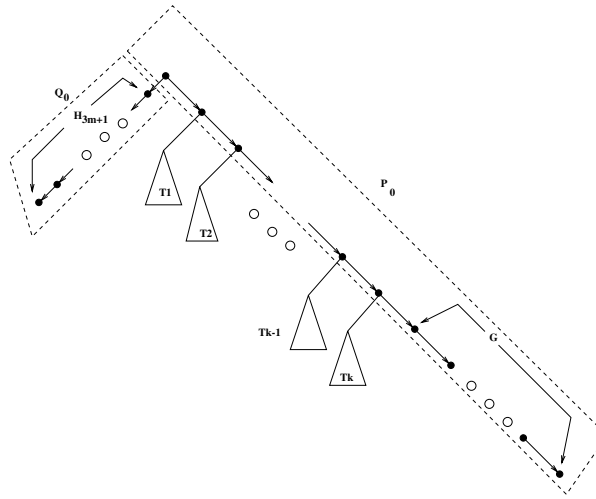


Fig. 2. *The tree $\mathcal{T}$ corresponding to the instance of X3C*

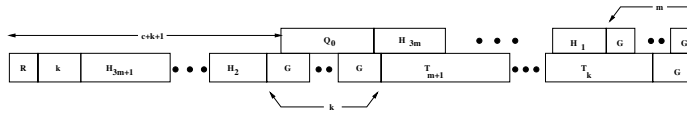


Fig. 3. *A feasible schedule (R represents the root of the $\mathcal{T}$, and k the first k tasks –after the root– of P_0)*

assigned to processor p_1. We proceed by proving a series of claims concerning such a schedule (due to lack of space the proofs of these claims will be given in the full version of the paper):
Claim 1: Communications can be afforded only from processor p_1 to p_2.
Claim 2: Processor p_2 may remain idle for at most k time units after time $c+1$.
Claim 3: All the tasks of every path P_i, $0 \leq i \leq k$ are executed by processor p_1. Therefore, p_2 may execute only tasks of Q_i's.
Claim 4: Processor p_2 executes at least $(H_{3m+1} - k)$ tasks of Q_0.
$a >> m$. Let us now call *stage* the time during which processor p_1 executes the tasks of a chain of length H_i of some P_t.
Claim 5: At every stage i, p_1 has to execute more tasks than at stage $i+1$.
Claim 6: Processor p_2 executes the tasks of exactly m Q_t's.

By Claim 5, in order to execute on p_2 $\sum_{i=1}^{3m+1} H_i + mG'$ tasks, X3C1 must have a solution F^* corresponding to the m Q_t's that processor p_2 has to execute. Thus we can state the following theorem.

Theorem 1. *Finding an optimal schedule for $P2 \mid$ binary trees, $\mathsf{p}_j = 1$, $c_{ij} = c \mid C_{max}$ is NP-hard in the strong sense.*

3 Polynomial Time Algorithm for Complete Trees

In this section, we prove that the problem becomes polynomial when the task graph is a *complete* binary out-tree (c.b.t.) T_h of height h (containing $n = 2^h - 1$ tasks). By convention we assume that the height of a leaf (resp. of the root) of the tree is one (resp. h) and that p_1 executes the root of the tree. By $M_{p_i},\ i = 1, 2$, we denote the last time unit some task is executed on processor p_i. A schedule is called *balanced* if $|M_{p_1} - M_{p_2}| \leq 1$.

The key point of an optimal schedule is the number of communications allowed in a path from the root of the tree to a leaf. Roughly speaking we distinguish three cases depending on the magnitude of c with respect to n:
- For "large" c, no communication is allowed (Lemma 2).
- For "medium" c, only one way communications, i.e. from p_1 to p_2, are allowed (Lemma 6).
- For "small" c, both ways communications, i.e. from p_1 to p_2 and from p_2 to p_1, are allowed (Lemma 3).

Lemma 2. *If $c \geq 2^h - 1 - h$, then the sequential schedule is optimal.*

Proof. Executing a task on p_2 introduces a communication delay on some path from the root to a leaf. The length of the schedule would be $M \geq h + c \geq n$. □

When $c < 2^h - 1 - h$ optimal schedules use both processors, i.e. tasks are executed also on p_2. We can derive a lower bound LB for the length of any schedule using both processors by considering a balanced schedule with the smallest number of idle slots (that is $c + 1$, since p_2 can start execution not before time $c + 1$):

$$C_{max}(\mathcal{S}) \geq LB = \left\lceil \frac{n + c + 1}{2} \right\rceil = \left\lceil \frac{2^h + c}{2} \right\rceil = 2^{h-1} + \left\lceil \frac{c}{2} \right\rceil .$$

In the following, by $x_h, x_{h-1}, ..., x_2, x_1$ we denote the leftmost path in a c.b.t. from its root, identified by x_h, to the leftmost leaf, identified by x_1. By y_i we denote the brother of task x_i, $1 \leq i \leq h - 1$, i.e. x_i and y_i are the left and the right child, respectively, of x_{i+1}.

Let us focus now on optimal schedules of length LB. If c is even, processor p_1 must execute tasks without interruption until time LB, as well as p_2 (starting at time $c + 1$). This is feasible for "small" even communication delays $c < 2^{h-2}$, by constructing a *two ways communication schedule.* i.e. a schedule where communications occur from p_1 to p_2 and from p_2 to p_1. The idea is to send one of the two subtrees of height $h - 1$ to p_2 immediately after the completion of the root on p_1, such that p_2 can start execution at time $c + 1$. Afterwards, in order to achieve the lower bound LB, several subtrees containing in total $c/2$ tasks are sent back from p_2 to p_1 (the schedule is now balanced).

The algorithm SchTwoWaysCom constructs such an optimal schedule:

procedure SchTwoWaysCom (*c.b.t. of height h, communication delay c even*)
begin
Choose the greatest value h_1, $1 \leq h_1 \leq h-2$, such that $c/2 \geq 2^{h_1} - 1$
Let $i = 1$, $M_{p_2} = c + 2^{h-1} - (2^{h_1} - 1)$ and $LB = \left\lceil \frac{2^h + c}{2} \right\rceil$
While $M_{p_2} > LB$ **do**
  Let $i = i + 1$
  Choose the greatest h_i, $1 \leq h_i \leq h_{i-1}$, such that $M_{p_2} - (2^{h_i} - 1) \geq LB$
  Let $M_{p_2} = M_{p_2} - (2^{h_i} - 1)$
enddo
Schedule on p_1 the tasks of the subtree rooted in y_{h-1}.
Schedule on p_2 the path $x_{h-1}, ..., x_{h_i+1}$ in consecutive time units from time $c+1$
Schedule on p_1 as soon as possible the subtrees rooted in y_{h_j}, $1 \leq j < i$.
If $h_i = h_{i-1}$ **then** schedule on p_1 as soon as possible the subtree rooted in x_{h_i}
  else schedule on p_1 as soon as possible the subtree rooted in y_{h_i}
Schedule on p_2 the rest tasks of the subtree rooted in x_{h-1}
end

Lemma 3. *The algorithm SchTwoWaysCom constructs an optimal schedule for even communication delays* $c \leq 2^{h-2} - 2$.

Proof. SchTwoWaysCom procedure aims to construct a schedule of length equal to LB. To this end, the subtree rooted in y_{h-1} is scheduled on p_1. The one rooted in x_{h-1} is sent on p_2, but some of its subtrees (of height $h_1, h_2, ..., h_i$) containing in total $c/2$ tasks, are sent back on p_1, otherwise p_2 ends execution c units of time later than p_1. Remark that if $h_i = h_{i-1}$, then the last two corresponding subtrees sent back to p_1 are rooted in $x_{h_i} = x_{h_{i-1}}$ and $y_{h_{i-1}}$.

Since 2^{h-1} is the last time unit occupied on p_1 by the root and the subtree rooted in y_{h-1} (executed in a non-idle manner), we have only to prove that the first subtree sent back from p_2 (rooted in y_{h_1}) is available on p_1 at time 2^{h-1}.

If the communication delay is $c = 2^{h-2} - 2$, the subtree rooted in y_{h_1} has height $h-3$, i.e. $h_1 = h-3$. Moreover, it is the only subtree ($i = 1$) sent back for execution on p_1. Since x_{h-2} is completed on p_2 at time $c+3$, y_{h-3} is available for execution on p_1 at time $2c+3$, and we have $2c+3 \leq 2^{h-1}$ that is true.

For shorter communication delays $c < 2^{h-2} - 2$ the same arguments hold. Subsequent trees (if any) sent back to p_1 arrive always before the completion of previous ones. □

Corollary 4. *The algorithm SchTwoWaysCom constructs a schedule of length at least* $LB + 1$ *for even communication delays* $c \geq 2^{h-2}$.

Corollary 5. *Any algorithm with two ways communication constructs a schedule of length at least* $LB + 1$ *for even communication delays* $c \geq 2^{h-2}$.

Proof. Any two ways communication algorithm either does not start execution on p_2 at time $c+1$ or has p_1 idle during at least one time unit (before executing the subtree rooted in y_{h_1}). □

We consider in the following the case of "medium" c's, i.e. $2^{h-2} \leq c < 2^h - 1 - h$. Since communication delays are too long, we construct a *one way communication schedule*, that is a schedule where communications occur only from p_1 to p_2, such that we never get two communications on some path from the root to a leaf. Now the idea is to send several subtrees to p_2 such that the resulting schedule is balanced and execution on p_2 starts as soon as possible.

The algorithm SchOneWayCom constructs such an optimal schedule:

procedure SchOneWayCom (*c.b.t. of height h, communication delay c*)
begin
Choose the greatest value h_1, $1 \leq h_1 \leq h-1$,
  such that $h - h_1 + c + 2^{h_1} - 1 \leq 2^h - 1 - (2^{h_1} - 1)$
Let $i = 1$, $M_{p_1} = 2^h - 1 - (2^{h_1} - 1)$ and $M_{p_2} = h - h_1 + c + 2^{h_1} - 1$
While $M_{p_1} > M_{p_2} + 1$ **do**
  Let $i = i + 1$
  Choose the greatest h_i, $1 \leq h_i \leq h_{i-1}$, s.t. $M_{p_2} + 2^{h_i} - 1 \leq M_{p_1} - (2^{h_i} - 1)$
  Let $M_{p_1} = M_{p_1} - (2^{h_i} - 1)$ and $M_{p_2} = M_{p_2} + 2^{h_i} - 1$
enddo
Schedule on p_1 the path $x_h, ..., x_{h_i+1}$ in the first $h - h_i$ time units.
Schedule on p_2 as soon as possible the subtrees rooted in y_{h_j}, $1 \leq j < i$.
If $h_i = h_{i-1}$ **then** schedule on p_2 as soon as possible the subtree rooted in x_{h_i}
  else schedule on p_2 as soon as possible the subtree rooted in y_{h_i}
Schedule on p_1 the rest tasks of the initial tree.
end

Remark. If $h_i = h_{i-1}$, then the corresponding subtrees sent to p_2 are rooted in $x_{h_i} = x_{h_{i-1}}$ and $y_{h_{i-1}}$.

Lemma 6. *The algorithm SchOneWayCom constructs an optimal schedule for:*
- *Odd communication delays $c < 2^h - 1 - h$, and*
- *Even communication delays $2^{h-2} \leq c < 2^h - 1 - h$.*

Proof. The length of the constructed schedule by algorithm SchOneWayCom is $M = \left\lceil \frac{2^h + c + h - h_1}{2} \right\rceil$. We distinguish between two cases depending on the magnitude of c:

(i) If $2^{h-1} - 1 \leq c < 2^h - 1 - h$ it is easy to observe that any two ways communication schedule is longer than the one way communication schedule produced by algorithm SchOneWayCom, $M_{SchTwoWaysCom} \geq h + 2c \geq \left\lceil \frac{2^h + c + h}{2} \right\rceil > M$, since there is at least one path in the tree containing two communications.
Thus, we have only to prove that the algorithm constructs the shortest schedule among all algorithms making one way communications. Consider a one way communication algorithm such that the first subtree to be executed on p_2 is of height $h'_1 > h_1$. Clearly the obtained schedule will be unbalanced and longer than the one produced by our algorithm. Consider now the case where the first subtree to be executed on p_2 is of height $h'_1 < h_1$. Then processor p_2 starts execution in time unit $h - h'_1 + c$ that is later than in algorithm SchOneWayCom and therefore the obtained schedule cannot be shorter.

(ii) If $c < 2^{h-1} - 1$ and c is odd, then the first subtree to be executed on p_2 is of height $h_1 = h - 2$. Hence, the length of the schedule is $\left\lceil \frac{2^h+c+1}{2} \right\rceil = LB$, that is optimal, since c is odd. If $2^{h-2} \leq c < 2^{h-1} - 1$ and c is even, then the length of the constructed schedule is equal to $LB + 1$. Using Corollary 5 we conclude that this schedule is optimal. □

Combining Lemmas 2, 3 and 6 we obtain the next theorem:

Theorem 7. *A complete binary tree of height h, with communication delay c, can be scheduled optimally on two processors in $O(n \log n)$ time.*

References

1. H. Ali, H. El-Rewini, *The time complexity of scheduling interval orders with communication is polynomial,* Parallel Processing Letters **3** (1) (1993) 53-58.
2. E. Bampis, A. Giannakos, J.-C. König, *On the complexity of scheduling with Large communication delays,* Europ. Journal of Operational Research **94** (1996) 252-260.
3. P. Chrétienne, C. Picouleau, *Scheduling with communication delays: A survey*, In *Scheduling Theory and Its Applications*, P. Chrétienne et al. (Eds.), J. Wiley, 1995.
4. E. G. Coffman Jr., R. L. Graham, *Optimal scheduling for two-processor systems*, Acta Informatica **1** (1972) 200-213.
5. L. Finta, Z. Liu, I. Milis, E. Bampis, *Scheduling UET-UCT series parallel graphs on two processors,* Theoretical Computer Science **162** (2) (1996) 323-340.
6. M. Fujii, T. Kasami, K. Ninomiya, *Optimal sequencing of two equivalent processors,* SIAM J. App. Math. **17** (4) (1969) 784-789.
7. L. Gao, A. L. Rosenberg and R. K. Sitaraman, *Optimal architecture-independent scheduling of fine-grain tree-sweep computations,* In Proc. 7th IEEE Symposium on Parallel and distributed Processing (1995) 620-629.
8. M.R. Garey, D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*, Ed. Freeman, 1979.
9. R. L. Graham, E. L. Lawler, J. K. Lenstra, K. Rinnooy Kan, *Optimization and approximation in deterministic scheduling: A survey*, Ann. Disc. Math., **5** (1979) 287-326.
10. A. Jakoby and R. Reischuk, *The complexity of scheduling problems with communication delays for trees,* In Proc. Scandinavian Workshop on Algorithm Theory, (SWAT'92), Springer Verlag LNCS-621 (1992) 165-177.
11. H. Jung, L. Kirousis, P. Spirakis, *Lower bounds and efficient algorithms for multiprocessor scheduling of DAGs with communication delays,* Information and Computation **105** (1993) 94-104.
12. J. K. Lenstra, M. Veldhorst and B. Veltman, *The complexity of scheduling trees with communication delays,* Journal of Algorithms, **20** (1) (1996) 157-173.
13. C. Papadimitriou, M. Yannakakis, *Towards an architecture-independent analysis of parallel algorithms,* SIAM J. on Computing, **2** (1990) 322-328.
14. T. Varvarigou, V. P. Roychowdhury, T. Kailath, E. Lawler, *Scheduling in and out forests in the presence of communication delays* IEEE Trans. on Parallel and Distributed Systems, **7** (10) (1996) 1065-1074.
15. B. Veltman, B. J. Lageweg, J. K. Lenstra, *Multiprocessor scheduling with communication delays*, Parallel Computing, **16** (1990) 173-182.

Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning*

Kirk Schloegel, George Karypis, and Vipin Kumar

Army HPC Research Center
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
(kirk, karypis, kumar)@cs.umn.edu

Abstract. Sequential multi-constraint graph partitioners have been developed to address the load balancing requirements of multi-phase simulations. The efficient execution of large multi-phase simulations on high performance parallel computers requires that the multi-constraint partitionings are computed in parallel. This paper presents a parallel formulation of a recently developed multi-constraint graph partitioning algorithm. We describe this algorithm and give experimental results conducted on a 128-processor Cray T3E. We show that our parallel algorithm is able to efficiently compute partitionings of similar edge-cuts as serial multi-constraint algorithms, and can scale to very large graphs. Our parallel multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E.

1 Introduction

Algorithms that find good partitionings of highly unstructured and irregular graphs are critical for the efficient execution of scientific simulations on high performance parallel computers. In these simulations, computation is performed iteratively on each element of a physical (2D or 3D) mesh, and then some information is exchanged between adjacent mesh elements. Efficient execution of these simulations requires a mapping of the computational mesh to the processors such that each processor gets a roughly equal number of elements and the amount of inter-processor communication required to exchange the information

* This work was supported by DOE contract number LLNL B347881, by NSF grant CCR-9972519, by Army Research Office contracts DA/DAAG55-98-1-0441, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Additional support was provided by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www-users.cs.umn.edu/~karypis

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 296–310, 2000.
© Springer-Verlag Berlin Heidelberg 2000

between adjacent mesh elements is minimized. This mapping is commonly found using a traditional graph partitioning algorithm. Even though the problem of graph partitioning is NP-complete, multilevel schemes [3, 7, 11, 12] have been developed that are able to quickly find excellent partitionings of graphs that correspond to the 2D or 3D irregular meshes used for scientific simulations.

Despite the success that multilevel graph partitioners have enjoyed, for many important classes of scientific simulations, the formulation of the traditional graph partitioning problem is inadequate. For example, in multi-phase simulations such as particle-in-mesh simulations, crash-worthiness testing, and combustion engine simulations, there exists synchronization steps between the different phases of the computation. The existence of these requires that each phase be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a decomposition based on this sum. Doing so may lead to some processors having too much work during one phase of the computation (and so, these may still be working after other processors are idle), and not enough work during another. Instead, it is critical that every processor have an equal amount of work from each of the phases of the computation. In general, multi-phase simulations require the partitioning to satisfy not just one, but a number of balance constraints equal to the number of computational phases.

Traditional graph partitioning techniques have been designed to balance a single computational phase only. An extension of the graph partitioning problem that can balance multiple phases is to assign a weight vector of size m to each vertex. The problem then becomes that of finding a partitioning that minimizes the total weight of the edges that are cut by the partitioning (i.e., the *edge-cut*) subject to the constraints that each of the m weights are balanced across the subdomains. Such a *multi-constraint* graph partitioning formulation as well as serial algorithms for computing multi-constraint partitionings are presented in [6].

It is desirable to compute multi-constraint partitionings in parallel for a number of reasons. Computational meshes in parallel scientific simulations are often too large to fit in the memory of one processor. A parallel partitioner can take advantage of the increased memory capacity of parallel machines. Thus, an effective parallel multi-constraint graph partitioner is key to the efficient execution of large multi-phase problems. Furthermore, in adaptive computations, the mesh needs to be partitioned frequently as the simulation progresses. In such computations, downloading the mesh to a single processor for repartitioning can become a major bottleneck.

The multi-constraint partitioning algorithm in [6] can be parallelized using the techniques presented in the parallel formulation of the single-constraint partitioning algorithm [8] as both are based on the multilevel paradigm. This paradigm consists of three phases: coarsening, initial partitioning, and multilevel refinement. In the coarsening phase, the original graph is successively coarsened down until it has only a small number of vertices. In the initial partitioning phase, a partitioning of the coarsest graph is computed. In the multilevel refine-

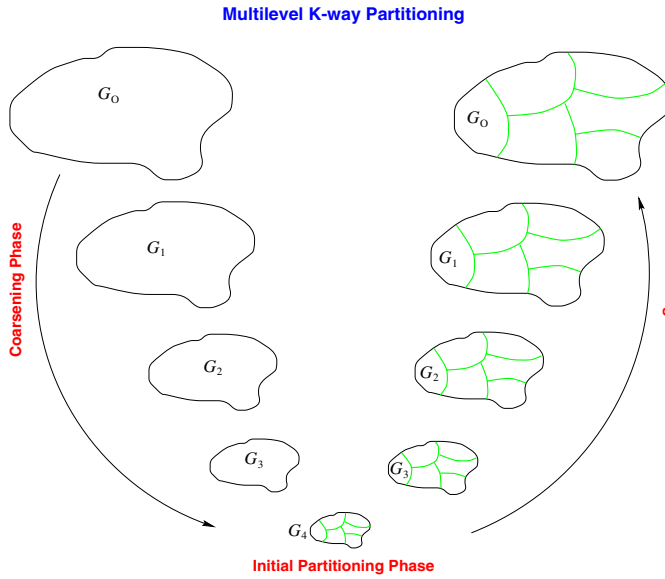


Fig. 1. The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the uncoarsening/refinement phase, the partitioning is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph.

ment phase, the initial partitioning is successively refined using a Kernighan-Lin (KL) type heuristic [10] as it is projected back to the original graph. Figure 1 illustrates the multilevel paradigm. Of these phases, it is straightforward to extend the parallel formulations of coarsening and initial partitioning to the context of multi-constraint partitioning. The key challenge is the parallel formulation of the refinement phase. The refinement phase for single-constraint partitioners is parallelized by relaxing the KL heuristic to the extent that the refinement can be performed in parallel while remaining effective. This relaxation can cause the partition to become unbalanced during the refinement process, but the imbalances are quickly corrected in succeeding iterations. Eventually, a balanced partitioning is obtained at the finest (i. e., the input) graph. Similar relaxation does not work for multi-constraint partitioning because it is non-trivial to correct load imbalances when more than one constraint is involved. A better approach is to avoid situations in which partitionings becomes imbalanced. This can be accomplished by either serializing the refinement algorithm, or else by restricting the amount of refinement that a processor is able to perform. The first will reduce the scalability of the algorithm and the second will result in low quality partitionings. Neither of these is desirable. Hence, the challenge in developing a parallel multi-constraint graph partitioner lies in developing a relaxation of the

refinement algorithm that is concurrent, effective, and maintains load balance for each constraint.

This paper describes a parallel multi-constraint refinement algorithm that is the key component of a parallel multi-constraint graph partitioner. We give experimental results of the full graph partitioning algorithm conducted on a 128-processor Cray T3E. We show that our parallel algorithm is able to compute balanced partitionings that have similar edge-cuts as those produced by the serial multi-constraint algorithm, while also being fast and scalable to very large graphs.

2 Parallel Multi-constraint Refinement

The main challenge in developing a parallel multi-constraint graph partitioner proved to be in developing a parallel multilevel refinement algorithm. This algorithm needs to meet the following criteria.

1. It must maintain the balance of all constraints.
2. It must maximize the possibility of refinement moves.
3. It must be scalable.

We briefly explain why developing an algorithm to meet all three of these criteria is challenging in the context of multiple constraints, and then describe our parallel multilevel refinement algorithm.

In order to guarantee that partition balance is maintained during parallel refinement, it is necessary to update global subdomain weights after every vertex migration. Such a scheme is much too serial in nature to be performed efficiently in parallel. For this reason, parallel single-constraint partitioning algorithms allow a number of vertex moves to occur concurrently before an update step is performed. One of the implications of concurrent refinement moves is that the balance constraint can be violated during refinement iterations. This is because if a subdomain can hold a certain amount of additional vertex weight without violating the balance constraint, then all of the processors assume that they can use all of this extra space for performing refinement moves. Of course, if just two processors move the amount of additional vertices that a subdomain can hold into it, then the subdomain will become overweight.

Parallel single-constraint graph partitioners address this challenge by encouraging subsequent refinement to restore the balance of the partitioning while improving its quality. For example, it is often sufficient to simply disallow further vertex moves into overweight subdomains and to perform another iteration of refinement. In general, the refinement process may not always be able to balance the partitioning while improving its quality in this way (although experience has shown that this usually works quite well). In this case, a few edge-cut increasing moves can be made to move vertices out of the overweight subdomains.

The real challenge is when we consider this phenomenon in the context of multiple balance constraints. This is because once a subdomain become overweight for a given constraint, it can be very difficult to balance the partitioning again.

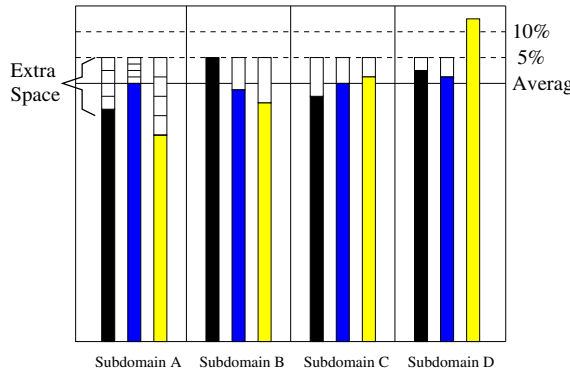


Fig. 2. This figure shows the subdomain weights for a 4-way partitioning of a 3-constraint graph. The white bars represent the extra space in a subdomain for each weight given a 5% user specified load imbalance tolerance.

Furthermore, the problem becomes more difficult as the number of constraints increases, as the multiple constraints are increasingly likely to interfere with each other during balancing. Given the difficulty of balancing multi-constraint partitionings, a better approach is to avoid situations in which the partitioning becomes imbalanced. Therefore, we would like to develop a multi-constraint refinement algorithm that can help to ensure that balance is maintained during parallel refinement.

One scheme that ensures that the balance is maintained during parallel refinement is to divide the amount of extra vertex weight that a subdomain can hold without becoming imbalanced by the number of processors. This then becomes the maximum vertex weight that any one processor is allowed to move into a particular subdomain in a single pass through the vertices. Consider the example illustrated in Figure 2. This shows the subdomain weights for a 4-way, 3-constraint partitioning. Lets assume that the user specified tolerance is 5%. The shaded bars represent the subdomain weights for each of the three constraints. The white bars represent the amount of weight that if added to the subdomain, would bring the bring the total weight to 5% above the average. In other words, the white bars show the amount of *extra space* each subdomain has for a particular weight given a 5% load imbalance tolerance. Figure 2 shows how the extra space in subdomain A can be split up for the four processors. If each processor is limited to moving the indicated amounts of weight into subdomain A, it is not possible for the 5% imbalance tolerance to be exceeded.

While this method guarantees that no subdomain (that is not overweight to start with) will become overweight beyond the imbalance tolerance, it is overly restrictive. This is because in general not all processors will need to use up their allocated space, while others may want to move more vertex weight into a subdomain than allowed by their slice. Furthermore, as the numbers of either processors or constraints increases, this effect increases. The reason is that as

the number of processors increases, the slices allocated to each processor get thinner. As the number of constraints increases, each additional constraint will also be sliced. This means that every vertex proposed for a move will be required to fit the slices of all of the constraints. For example, consider a three-constraint, ten-way partitioning computed on ten processors. If subdomain A can hold 20 units of the first weight, 30 units of the second weight, and 10 units of the third weight, then every processor must ensure that the sum of the weight vectors of all of the vertices that it moves into subdomain A is less than (2, 3, 1). It could very easily be the case then that this restriction is too severe to allow any one processor to perform their desired refinement.

It is possible to allocate the extra space of the subdomains more intelligently than simply giving each processor an equal share. We have investigated schemes that make the allocations based on a number of factors, such as the potential edge-cut improvements of the border vertices from a specific processor to a specific subdomain, the weights of these border vertices, and the total number of border vertices on each processor. While these schemes allow a greater number of refinement moves to be made than the straightforward scheme, they still restrict more edge-cut reducing moves than the serial algorithm. Our experiments have shown that these schemes produce partitionings that are up to 50% worse in quality than the serial multi-constraint algorithm. (Note, these results are not presented in this paper.)

Our Parallel Multi-constraint Refinement Algorithm. We have developed a parallel multi-constraint refinement algorithm that is no more restrictive than the serial algorithm with respect to the number of refinement moves that it allows, while also helping to ensure that none of the constraints become overly imbalanced. In the multilevel context, this algorithm is just as effective in improving the edge-cuts of partitionings as the serial algorithm.

This algorithm (essentially a reservation scheme) performs an additional pass through the vertices on every refinement iteration. In the first pass, refinement moves are made concurrently (as normal), however, only temporary data structures are updated. Next, a global reduction operation is performed to determine whether or not the balance constraints will be violated if these moves commit. If none of the balance constraints are violated, the moves are committed as normal. Otherwise, each processor is required to disallow a portion[1] of its proposed vertex moves into those subdomains that would be overweight if all of the moves are allowed to commit. The specific moves to be disallowed are selected randomly by each processor. While selecting moves randomly can negatively impact the edge-cut, this is usually not a problem because further refinement can easily correct the effects of any poor selections that happen to be made. Except for these modifications, our multi-constraint refinement algorithm is similar to the coarse-grain refinement algorithm described in [4].

[1] This portion is equal to one minus the amount of extra space in the subdomain divided by the total weight of all of the proposed moves into the subdomain.

It is important to note that the above scheme does not guarantee that the balance constraints will be maintained. This is because when we disallow a number of vertex moves, the weights of the subdomains from which these vertices were to have moved become higher than the weights that had been computed with the global reduction operation. It is therefore possible for some of these subdomains to become overweight. To correct this situation, a second global reduction operation can be performed followed by another round in which a number of the (remaining) proposed vertex moves are disallowed. These corrections might then lead to other imbalances, whose corrections might lead to others, and so on. We can easily allow this process to iterate until it converges (or until all of the proposed moves have been disallowed). Instead, we have chosen to simply ignores this problem. This is because the number of disallowed moves is a very small fraction of the total number of vertex moves, and so, any imbalance that is brought about by them is quite modest. Our experimental results show that the amount of imbalance introduced in this way is small enough that further refinement is able to correct it. In fact, as long as the amount of imbalance introduced is correctable, such a scheme can potentially result in higher quality partitionings compared to schemes that explore the feasible solution space only. (See the discussion in Section 3.)

Scalability Analysis. The scalability analysis of a parallel multilevel (single-constraint) graph partitioner is presented in [8]. This analysis assumes that (i) each vertex in the graph has a small bounded degree, (ii) this property is also satisfied by the successive coarser graphs, and (iii) the number of nodes in successive coarser graphs decreases by a factor of $1 + \epsilon$, where $0 < \epsilon \leq 1$. (Note, these assumptions hold true for all graphs that correspond to well-shaped finite element meshes.) Under these assumptions, the parallel run time of the single-constraint algorithm is

$$T_{par} = O\left(\frac{n}{p}\right) + O(p \log n) \tag{1}$$

and the isoefficiency function is $O(p^2 \log p)$, where n is the number of vertices and p is the number of processors. The parallel run time of our multi-constraint graph partitioner is similar (given the two assumptions). However, during both graph coarsening and multilevel refinement, all m weights must be considered. Therefore, the parallel run time of the multi-constraint algorithm is m times longer, or

$$T_{par} = O\left(\frac{nm}{p}\right) + O(pm \log n). \tag{2}$$

Since the sequential complexity of the serial multi-constraint algorithm is $O(nm)$, the isoefficiency function of the multi-constraint partitioner is also $O(p^2 \log p)$.

3 Experimental Results

In this section, we present experimental results of our parallel multi-constraint k-way graph partitioning algorithm on 32, 64, and 128 processors of a Cray T3E.

We constructed two sets of test problems to evaluate the effectiveness of our parallel partitioning algorithm in computing high-quality, balanced partitionings quickly. Both sets of problems were generated synthetically from the four graphs described in Table 1.

The purpose of the first set of problems is to test the ability of the multi-constraint partitioner to compute a balanced k-way partitioning for some relatively hard problems. From each of the four input graphs we generated graphs with two, three, four, and five weights, respectively. For each graph, the weights of the vertices were generated as follows. First, we computed a 16-way partitioning of the graph and then we assigned the same weight vector to all of the vertices in each one of these 16 subdomains. The weight vector for each subdomain was generated randomly, such that each vector contains m (for $m = 2, 3, 4, 5$) random numbers ranging from 0 to 19. Note that if we do not compute a 16-way partitioning, but instead simply assign randomly generated weights to each of the vertices, then the problem reduces to that of a single-constraint partitioning problem. The reason is that due to the random distribution of vertex weights, if we select any l vertices, the sum of their weight vectors will be around (lr, lr, ..., lr) where r is the expected average value of the random distribution. So the weight vector sums of any two sets of l vertices will tend to be similar regardless of the number of constraints. Thus, all we need to do to balance m constraints is to ensure that the subdomains contain a roughly equal number of vertices. This is the formulation for the single-constraint partitioning problem. Requiring that all of the vertices within a subdomain have the same weight vector avoids this effect. It also better models many applications. For example, in multi-phase problems, different regions of the mesh are active during different phases of the computation. However, those mesh elements that are active in the same phase typically form groups of contiguous regions and are not distributed randomly throughout the mesh. Therefore, each of the 16 subdomains in the first problem set models a contiguous region of mesh elements.

The purpose of the second set of problems is to test the performance of the multi-constraint partitioner in the context of multi-phase computations in which different (possibly overlapping) subsets of nodes participate in different phases. For each of the four graphs, we again generated graphs with two, three, four, and five weights corresponding to a two-, three-, four-, and five-phase computation, respectively. In the case of the five-phase graph, the portion of the graph that is active (i.e., performing computations) is 100%, 75%, 50%, 50%, and 25% of the subdomains. In the four-phase case, this is 100%, 75%, 50%, and 50%. In the three- and two-phase cases, it is 100%, 75%, and 50% and 100% and 75%, respectively. The portions of the graph that are active was determined as follows. First, we computed a 32-way partitioning of the graph and then we randomly selected a subset of these subdomains according to the overall active percentage. For instance, to determine the portion of the graph that is active during the second phase, we randomly selected 24 out of these 32 subdomains (i.e., 75%). The weight vectors associated with each vertex depends on the phases in which it is active. For instance, in the case of the five-phase computation, if a vertex

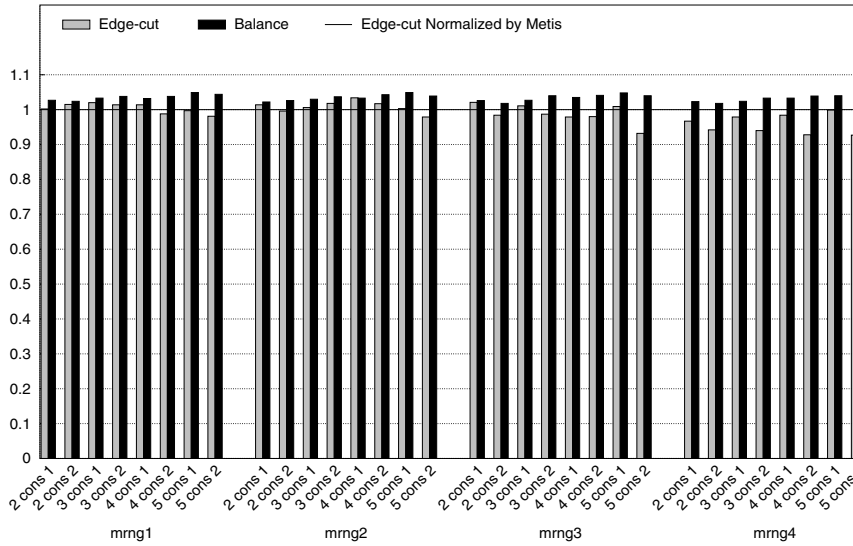


Fig. 3. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 32 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in MeTiS.

is active only during the first, second, and fifth phase, its weight vector will be (1, 1, 0, 0, 1). In generating these test problems we also assigned weight to the edges to better reflect the overall communication volume of the underlying multi-phase computation. In particular, the weight of an edge (v, u) was set to the number of phases that both vertices v and u are active at the same time. This is an accurate model of the overall information exchange between vertices since during each phase, vertices access each other's data only if both are active.

Graph	Num of Verts	Num of Edges
$mrng1$	257,000	1,010,096
$mrng2$	1,017,253	4,031,428
$mrng3$	4,039,160	16,033,696
$mrng4$	7,533,224	29,982,560

Table 1. Characteristics of the various graphs used in the experiments.

Comparison of Serial and Parallel Multi-constraint Algorithms. Figures 3, 4, and 5 compare the edge-cuts of the partitionings produced by our parallel multi-constraint graph partitioning algorithm with those produced by the serial multi-

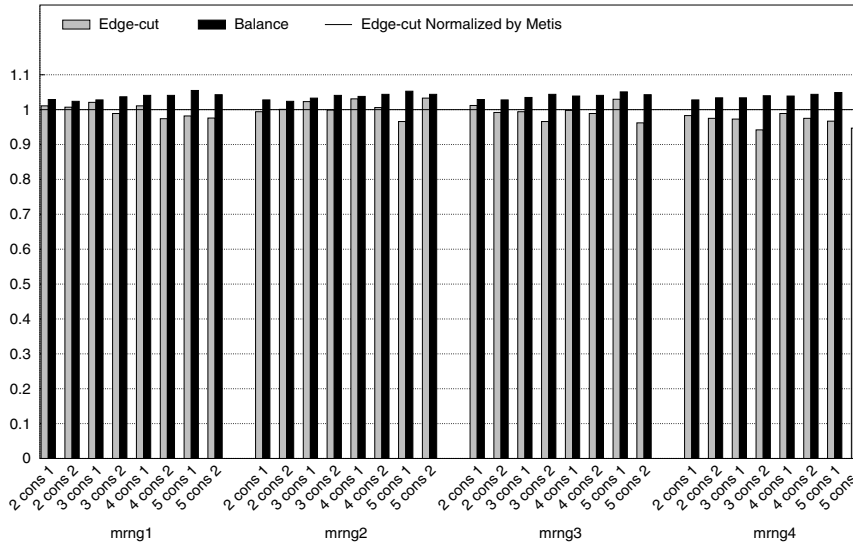


Fig. 4. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 64 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in MéTiS.

constraint algorithm [6], and give the maximum load imbalance of the partitionings produced by our algorithm. Each figure shows four sets of results, one for each of the four graphs described in Table 1. Each set is composed of two-, three-, four-, and five-constraint Type 1 and 2 problems. These are labeled "m cons t" where m indicates the number of constraints and t indicates the type of problem (i.e., Type 1 or 2). So the results labeled "2 cons 1" indicates the edge-cut and balance results for a two-constraint Type 1 problem. The edge-cut results shown are those obtained by our parallel algorithm normalized by those obtained by the serial algorithm. Therefore, a bar below the 1.0 index line indicates that our parallel algorithm produced partitionings with better edge-cuts than the serial algorithm. The balance results indicate the maximum imbalance of all of the constraints. (Here, imbalance is defined as the maximum subdomain weight divided by the average subdomain weight for a given constraint.) These results are not normalized. Note that we set an imbalance tolerance of 5% for all of the constraints. The results given in Figures 3, 4, and 5 give the arithmetic means of three runs by our algorithm utilizing different random seeds each time. Note that in every case, the results of each individual run were within a few percent of the averaged results shown. For each figure, the number of subdomains computed is equal to the number of processors.

Figures 3, 4, and 5 show that our parallel multi-constraint graph partitioning algorithm is able to compute partitionings with similar or better edge-cuts

compared to the serial multi-constraint graph partitioner, while ensuring that multiple constraints are balanced.

Notice that the parallel algorithm is sometimes able to produce partitionings with better edge-cuts than the serial algorithm. There are two reasons for this. First, the parallel formulation of the matching scheme used (heavy-edge matching using the balanced-edge heuristic as a tie-breaker [6]) is not as effective in finding a maximal matching as the serial formulation. (This is due to the protocol that is used to arbitrate between conflicting matching requests made in parallel [4].) Therefore, a smaller number of vertices match together with the parallel algorithm than with the serial algorithm. The result is that the newly computed coarsened graph tends to be larger for the parallel algorithm than for the serial algorithm, and so, the parallel algorithm takes more coarsening levels to obtain a sufficiently small graph. The effect of this is that the matching algorithm usually has one or more additional coarsening levels in which to remove exposed edge weight (i. e., the total weight of the edges on the graph). By the time the parallel algorithm computes the coarsest graph, it can have significantly less exposed edge weight than the coarsest graph computed by the serial algorithm. This makes it easier for the initial partitioning algorithm to compute higher-quality partitionings. During multilevel refinement, some of this advantage is maintained, and so, the final partitioning can be better than those computed by the serial algorithm. The disadvantage of slow coarsening is that the additional coarsening and refinement levels take time, and so, the execution time of the algorithm is increased. This phenomenon of slow coarsening was also observed in the context of hypergraphs in [1].

The second reason is that in the serial algorithm, once the partitioning becomes balanced it will never explore the infeasible solution space in order to improve the edge-cut. Since the parallel refinement algorithm does not guarantee to maintain partition balance, the parallel graph partitioner may do so. This usually happens on the coarse graphs. Here, the granularity of the vertices makes it more likely that the parallel multi-constraint refinement algorithm will result in slightly imbalanced partitionings. Essentially, the parallel multi-constraint refinement algorithm it too aggressive in reducing the edge-cut here, and so, makes too many refinement moves. This is a poor strategy if the partitioning becomes so imbalanced that subsequent refinement is not able to restore the balance. However, since our parallel refinement algorithm helps to ensure that the amount of imbalance introduced is small, subsequent refinement is able to restore the partition balance while further improving its edge-cut.

Run Time Results. Table 2 compares the run times of the parallel multi-constraint graph partitioning algorithm with the serial multi-constraint algorithm implemented in the METIS library [5] for *mrng*1. These results show only modest speedups for the parallel partitioner. The reason is that the graph *mrng*1 is quite small, and so, the communication and parallel overheads are significant. However, we use *mrng*1 because it is the only one of the test graphs that is small enough to run serially on a single processor of the Cray T3E.

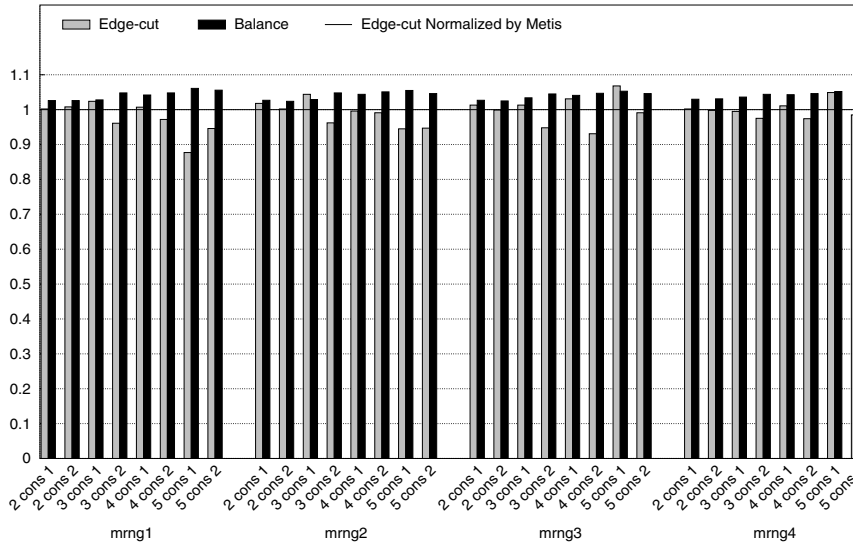


Fig. 5. This figure shows the edge-cut and balance results from the parallel multi-constraint algorithm on 128 processors. The edge-cut results are normalized by the results obtained from the serial multi-constraint algorithm implemented in MéTiS.

Table 3 gives selected run time results and efficiencies of our parallel multi-constraint graph partitioning algorithm on up to 128 processors. Table 3 shows that our algorithm is very fast, as it is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E. It also shows that our parallel algorithm obtains similar run times as you double (or quadruple) both the size of problem and the number of processors. For example, the time required to partition $mrng2$ (with 1 million vertices) on eight processors is similar to that of partitioning $mrng3$ (4 million vertices) on 32 processors and $mrng4$ (7.5 million vertices) on 64 processors.

k	serial time	parallel time
2	7.3	6.4
4	7.5	4.4
8	8.0	2.5
16	8.3	1.7

Table 2. Serial and parallel run times of the multi-constraint graph partitioner for a three-constraint problem on $mrng1$.

Graph	8-processors		16-processors		32-processors		64-processors		128-processors	
	time	efficiency	time	efficiency	time	efficiency	time	efficiency	time	efficiency
*mrng*2	9.8	100%	5.3	92%	3.5	70%	2.5	49%	3.1	20%
*mrng*3	31.8	100%	16.9	94%	9.3	85%	5.7	70%	4.4	45%
*mrng*4	out of mem.		30.7	100%	16.7	92%	9.2	83%	6.4	60%

Table 3. Parallel run times and efficiencies of our multi-constraint graph partitioner on three-constraint type 1 problems.

Graph	8-processors	16-processors	32-processors	64-processors	128-processors
*mrng*2	5.4	3.1	2.1	1.5	1.7
*mrng*3	15.8	8.8	4.8	3.0	2.7
*mrng*4	38.6	16.2	8.8	5.0	3.6

Table 4. Parallel run times of the single-constraint graph partitioner implemented in ParMeTiS.

Table 4 gives the run times of the k-way single-constraint parallel graph partitioning algorithm implemented in the ParMeTiS library [9] on the same graphs used for our experiments. Comparing Tables 3 and 4 shows that computing a three-constraint partitioning takes about twice as long as computing a single-constraint partitioning. For example, it takes 9.3 seconds to compute a three-constraint partitioning and 4.8 seconds to compute a single-constraint partitioning for *mrng*3 on 32 processors. Also, comparing the speedups indicates that the multi-constraint algorithm is slightly more scalable than the single-constraint algorithm. For example, the speedup from 16 to 128 processors for *mrng*3 is 3.84 for the multi-constraint algorithm and 3.26 for the single-constraint algorithm. The reason is that the multi-constraint algorithm is more computationally intensive than the single-constraint algorithm, as multiple (not single) weights must be computed regularly.

Parallel Efficiency. Table 3 gives selected parallel efficiencies of our parallel multi-constraint graph partitioning algorithm on up to 128 processors. The efficiencies are computed with respect to the smallest number of processors shown. Therefore, for *mrng*2 and *mrng*3, we set the efficiency of eight processors to 100%, while we set the efficiency of 16 processors to 100% for *mrng*4. The parallel multi-constraint graph partitioner obtained efficiencies between 20% and 94%. The efficiencies were good (between 90% - 70%) when the graph was sufficiently large with respect to the number of processors. However, these dropped off for the smaller graphs on large number of processors. The isoefficiency of the parallel multi-constraint graph partitioner is $O(p^2 \log p)$. Therefore, in order to

maintain a constant efficiency when doubling the number of processors, we need to increase the size of the data by a little more than four times. Since $mrng3$ is approximately four times as large as $mrng2$ we can test the isoefficiency function experimentally. The efficiency of the multi-constraint partitioner with 32 processors for $mrng2$ is 70%. Doubling the number of processors to 64 and increasing the data size by four times (64-processors on $mrng3$) yields a similar efficiency. This is better than expected, as the isoefficiency function predicts that we need in increase the size of the data set by more than four times to obtain the same efficiency. If we examine the results of 64 processors on $mrng2$ and 128 processors on $mrng3$ we see a slightly decreasing efficiency of 49% to 45%. This is what we would expect based on the isoefficiency function. If we examine the results of 16 processors on $mrng2$ and 32 processors on $mrng3$ we see that the drop in efficiency is larger (92% to 85%). So here we get a slightly worse efficiency than expected. These experimental results are quite consistent with the isoefficiency function of the algorithm. The slight deviations can be attributed to the fact that the number of refinement iterations on each graph is upper bounded. However, if a local minima is reached prior to this upper bound, then no further iterations will be performed on this graph. Therefore, while the upper bound on the amount of work done by the algorithm is the same for all of the experiments, the actual amount of work done can be slightly different depending on the refinement process.

4 Conclusions

This paper has presented a parallel formulation of the multi-constraint graph partitioning algorithm for partitioning 2D and 3D irregular and unstructured meshes used in scientific simulations. This algorithm is essentially as scalable as the widely used parallel formulation of the single-constraint graph partitioning algorithm [8]. Experimental results conducted on a 128-processor Cray T3E show that our parallel algorithm is able to compute balanced partitionings with similar edge-cuts as the serial algorithm. We have shown that the run time of our algorithm is very fast. Our parallel multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7.5 million node graph in about 7 seconds on 128 processors of a Cray T3E.

Although the experiments presented in this paper are all conducted on synthetic graphs, our parallel multi-constraint partitioning algorithm has also been tested on real application graphs. Basermann et al. [2] have used the parallel multi-constraint graph partitioner described in this paper for load balancing multi-phase car crash simulations of an Audi and a BMW in frontal impacts with a wall. These results are consistent with the run time, edge-cut, and balance results presented in Section 3.

While the experimental results presented in Section 3 (and [2]) are quite good, it is important to note that the effectiveness of the algorithm depends on two things. First, it is critical that a relatively balanced partitioning be computed during the initial partitioning phase. This is because if the partitioning starts out

imbalanced, there is no guarantee that it will ever become balanced during the course of multilevel refinement. Our experiments (not presented in this paper) have shown that an initial partitioning that is more than 20% imbalanced for one or more constraints is unlikely to be improved during multilevel refinement. Second, as is the case for the serial multi-constraint algorithm, the quality of the final partitioning is largely dependent on the availability of vertices that can be swapped across subdomains in order to reduce the edge-cut, while maintaining all of the balance constraints. Experimentation has shown that for a small number of constraints (i.e., two to four) there is a good availability of such vertices, and so, the quality of the computed partitionings is good. However, as the number of constraints increases further, the number of vertices that can be moved while maintaining all of the balance constraints decreases. Therefore, the quality of the produced partitionings can drop off dramatically.

References

[1] C. Alpert, J. Huang, and A. Kahng. Multilevel circuit partitioning. In *Proc. of the 34th ACM/IEEE Design Automation Conference*, 1997.

[2] A. Basermann, J. Fingberg, G. Lonsdale, B. Maerten, and C. Walshaw. Dynamic multi-partitioning for parallel finite element applications. *Submitted to ParCo '99*, 1999.

[3] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. *Proceedings Supercomputing '95*, 1995.

[4] G. Karypis and V. Kumar. A coarse-grain parallel multilevel k-way partitioning algorithm. In *Proceedings of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[5] G. Karypis and V. Kumar. MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. Technical report, Univ. of MN, Dept. of Computer Sci. and Engr., 1998.

[6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing '98*, 1998.

[7] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[8] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[9] G. Karypis, K. Schloegel, and V. Kumar. ParMeTiS: Parallel graph partitioning and sparse matrix ordering library. Technical report, Univ. of MN, Dept. of Computer Sci. and Engr., 1997.

[10] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[11] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. Technical report, University of Paderborn, 1999.

[12] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical Report 99/IM/44, University of Greenwich, UK, 1999.

Experiments with Scheduling Divisible Tasks in Clusters of Workstations

Maciej Drozdowski[1]* and Paweł Wolniewicz[2]

[1] Institute of Computing Science, Poznań University of Technology, ul.Piotrowo 3a, 60-965 Poznań, Poland.
[2] Poznań Supercomputing and Networking Center, ul.Noskowskiego 10, 61-794 Poznań, Poland.

Abstract. We present results of a series of experiments with parallel processing divisible tasks on various cluster of workstations platforms. Divisible task is a new model of scheduling distributed computations. It is assumed that the parallel application can be divided into parts of arbitrary sizes and the parts can be processed independently on distributed computers. Though practical verification of the scheduling model was the primary goal of the experiments also an insight into the behavior and performance of cluster computing platforms has been gained.

Keywords: Scheduling, divisible tasks, clusters of workstations.

1 Introduction

The first work analyzing divisible tasks [3] was motivated by the need of finding optimal balance between parallelism of computations and necessary communication in a linear network of intelligent sensors. Later divisible task model was used to represent distributed computations in: linear arrays of processors, stars, buses and trees of processors, hypercubes, meshes, multistage interconnections [1,4]. It was demonstrated that divisible task theory can be a useful tool in performance evaluation of distributed computations. Experiments performed in a dedicated Transputer system [2] confirm correctness of the theory predictions. This work is dedicated to verifying divisible task model in contemporary parallel processing environments available to the masses.

The remaining parts of this paper are organized as follows. In the next section we formulate the problem of scheduling divisible task in star networks. In Section 3 we describe test applications. In Section 4 the way of experimenting and the results obtained are presented. In Section 5 the results are discussed and conclusions are proposed.

* The research has been partially supported by a KBN grant and project CRIT2. Corresponding author.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 311–319, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Processing Divisible Tasks on Star and Bus Topologies

In the divisible task model it is assumed that computations (or work, load, processing) can be divided into several parts of arbitrary sizes which can be processed on parallel processors. In other words, granularity of parallelism is fine because the work can be divided into chunks of arbitrary sizes. There are no precedence constraints (or data dependencies) because the parts can be processed independently on parallel processors. Applications conforming with divisible task model are e.g. distributed search for a pattern in text, audio, graphical, and database files; distributed processing of big measurement data files; many problems of simulation, linear algebra and combinatorial optimization.

We assume that initially whole volume V of work that must be performed (or e.g. data to be processed) resides on one processor called *originator*. In the star (equivalently bus) interconnection the originator activates other processors one after another by sending them some amount of load for processing. It is assumed that the load is sent to the processors only once. α_i denotes the amount sent to processor P_i, for $i = 1, \ldots, m$. The transmission time is equal to $S_i + \alpha_i C_i$, where S_i is communication startup time, and C_i is transfer rate. The time of processing α_i units of work on P_i is $\alpha_i A_i$. The units of V, S_i, C_i, and A_i can be e.g. bytes, seconds, and seconds per byte (twice), respectively. Communication rates and startup times can represent not only the network hardware but also all the layers of communication software between the user application modules. Having received its share of work, each of the computers immediately starts processing it and finally returns to the originator the results in the amount of $\beta(\alpha_i)$. $\beta(x)$ is an application-dependent function of the amount of results produced per x units of input data. In Fig.1 a Gantt chart with communications and computations in the star network is presented. In Fig.1a results are returned in the inverted order of activating the processors (which we will call LIFO), and in Fig.1b in the same order in which processors obtained their work (FIFO).

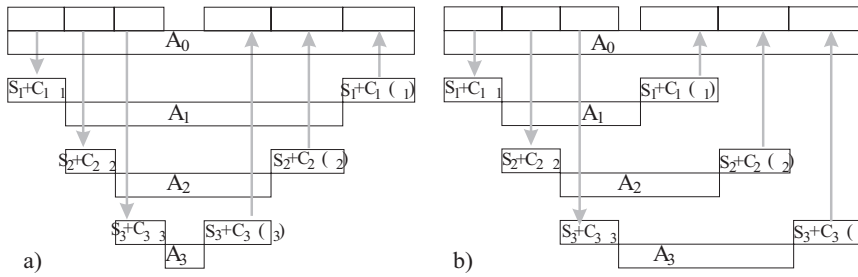


Fig. 1. Communications and computations in star. a) LIFO case, b) FIFO case.

Our goal is to distribute computations, i.e. find α_i, such that the duration of all communications and computations is minimal. Observe (cf. Fig.1a) that in the LIFO case processing on the processor activated earlier lasts as long as sending

to the next processor, computing on it and returning the results. Using this observation we can formulate a set of linear equations from which distribution of the load can be found:

$$\alpha_i A_i = 2S_{i+1} + C_{i+1}(\alpha_{i+1} + \beta(\alpha_{i+1})) + A_{i+1}\alpha_{i+1} \quad i=0,\ldots,m-1 \tag{1}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{2}$$

P_0 denotes the originator. In the FIFO case (cf. Fig.1b) the time of processing on P_i and returning results from processor P_i is equal to the time of sending to P_{i+1} and processing on P_{i+1}. Hence, distribution of the work can be calculated from equations:

$$\alpha_i A_i + S_i + \beta(\alpha_i)C_i = S_{i+1} + \alpha_{i+1}(C_{i+1} + A_{i+1}), \quad i=1,\ldots,m-1 \tag{3}$$

$$\alpha_0 A_0 = \sum_{i=1}^{m}(S_i + \alpha_i C_i) + \alpha_m A_m + S_m + C_m\beta(\alpha_m) \tag{4}$$

$$V = \sum_{i=0}^{m} \alpha_i \tag{5}$$

Due to specific structure the above two equation systems can be solved in $O(m)$ time. However, they may have no feasible solution (because some $\alpha_i < 0$) when volume V is too small and not all m processors are able to take part in the computation. In this case less processors should be used.

3 Test Applications

3.1 Search for a Pattern

The problem consists in verifying whether some given sequence S of characters contains substring x. If it is the case the position of the first character in S matching x is returned as a result. Having calculated quantity α_i of data the originator sends to processor P_i amount of $\alpha_i + \texttt{strlen}(x) - 1$ bytes from the sequence S starting at position $\sum_{j=0}^{i-1} \alpha_j + 1$. The chunks overlap in order to avoid cutting substring x placed across the border of two different chunks. As the files for the tests were known the amount of returned results was also known. $\beta(x) \approx 0.005x$ which is typical of search in databases holding personal data.

3.2 Compression

In this application originator sends to the processors parts of a file. The part sent to processor P_i has size α_i. Each of the processors compresses the obtained data using LZW compression algorithm. The resulting compressed strings are returned to the originator and appended to one output file. The original file can be obtained by decompressing each part in turn. The achieved compression ratio

determines the amount of returned results. It was measured that $\beta(x) = 0.55x$. The compression ratio and speed depend on the contents and size of the input. In order to eliminate (or at least minimize) this dependence only parts of at least 10kB were sent to the processors for remote compression.

3.3 Join

Join is a fundamental operation in relational databases. Suppose there are two databases: A e.g. with a list of supplier identifiers, names, addresses etc., and B with a list of products with names, prices, etc. and supplier identifier. The result of join operation on A and B should be one file with a list of suppliers (names, addresses, etc.) and the products the respective supplier provides. The join algorithm can be understood as calculation of cartesian product $A \times B$ of the two initial databases. $A \times B$ can be viewed as a 2-dimensional array in which one row corresponds to one record a_j from file A and one column corresponds to one record b_k from database B. On the intersection of row a_j and column b_k pair (a_j, b_k) is created which is transferred to the output file only if the fields of the supplier identifier match. In our implementation of distributed join, one of the databases (say A) was transmitted to all processors first. Then, the second database (B) was cut into parts B_i according to the calculated voulmes α_i, and sent to processors $P_i\,(i = 1, \ldots, m)$. Each of the processors calculated join on A and B_i, and returned the results to the originator. Databases A and B were artificially and randomly generated, therefore the amount of results was known. β expressed the ratio of the amount of the results and database B size.

3.4 Graph Coloring and Genetic Search

Consider graph $G(V, E)$, where V is a set of vertices, and $E = \{\{v_i, v_j\} : v_i, v_j \in V\}$ is a set of edges. Node coloring problem consists in assigning colors to the nodes such that no two adjacent nodes v_i, v_j have the same color. More precisely, node coloring is a mapping $f : V \rightarrow \{1, \ldots, k\}$, where $\{v_i, v_j\} \in E \Rightarrow f(v_i) /= f(v_j)$. Find minimum k, i.e. chromatic number χ_G.

Determining chromatic number is a hard combinatorial problem, therefore genetic search metaheuristics was used to solve it approximately. In our implementation of the genetic search each solution is a gene represented by a string of colors assigned to the consecutive nodes. Good solutions from the initial population are combined using genetic operators to obtain a new population. The measure of solution quality is called fitness function which in our case was the number of the colors used plus the number of infeasibly colored nodes. Two genetic operators were used to obtain new 'individuals': crossover and mutation. Crossover is a binary operator exchanging tails of the strings in two genes starting at a randomly selected place. Mutation operator makes random changes in the individuals and diversifies the population. Solutions were selected to produce offspring with probability increasing proportionally to decreasing of the fitness function (note that we have minimization). Originator generated initial population of 1000 random solutions. This population was distributed among the

processors according to the calculated values of α_i's. Each processor created a fixed number of new generations and returned final population to the originator. Thus, $\beta(x) = x$. A feasible solution with the smallest number of used colors was the final outcome.

4 The Results

In this section we outline results of the experiments. We examined several different hardware and software platforms. Due to time and workforce limitations not all applications were performed on every considered platform. In Table 1 we summarize which application was tested on which platform. All experiments were made on Ethernet network. Abbreviation ded. stands for dedicated single-segment interconnection, and pub. for public, multisegment network.

Table 1. Platforms vs applications

application→ (year)platform	search for a pattern	com-pression	join	coloring
A: (1995) 7 heterogeneous Sun workstations: SLC, IPX, SparcClassic, PVM, ded.10Mb	yes			
B: (1997) 6 heterogeneous PCs: 486DX66, RAM 8M - P166, RAM 64M, Linux, PVM, pub.10Mb		yes		
C: (1997) 7 nodes of IBM SP2, PVM, ded.45Mb		yes		
D: (1999) 6 homogeneous PCs: P-133, RAM 64M, WinNT, MPI, ded.100Mb	yes	yes	yes	
E: (1999) 4 heterogeneous PCs:P-100, RAM 24M - Celeron-330, RAM 64M, Win98, Java, pub.10Mb				yes
F: (1999) 6 homogeneous PCs: P-200MMX, RAM 32M, Linux, Java, ded.100Mb				yes

The main goal of the experiments was to apply divisible task model in practice and to verify correctness of its predictions. The verification was done by comparing the real and the predicted execution times of some application when data is distributed in chunks of sizes (α_i's) calculated from equations (1)-(2) or (3)-(5). To formulate the above equations we needed parameters A_j, C_j, S_j for $j = 1, \ldots, m$. Therefore, we had to measure these parameters first. The communication parameters were measured by a ping-pong test. Originator sent to a processor some amount of data. The processor immediately returned these data. A symmetry of the communication links was assumed and half of the total bidirectional communication time was taken as the unidirectional communication time. The communication time and the amount of data were stored. After collecting a number of such pairs (for various sizes of the message), parameters S_j, C_j were calculated using linear regression. Processing rate A_j was measured as an average of the ratios of the computation time and the quantity of data processed. The method of obtaining $\beta(x)$ has been explained in the previous section. The

measured communication parameters are presented in Table 2. Standard deviations are reported after the ± sign. The last two rows apply to the same hardware suite as for platform F. These data were obtained in some other set of experiments. Table 2 requires some comment and explanation. Firstly, these numbers

Table 2. Typical values of communication parameters

platform	C_j[μs/B]	S_j[μs]
A: various Sun workstations, PVM, ded.10Mb	70.7±0.3	636000±86000
B: various PCs, Linux, PVM, pub.10Mb	7031±13	2861±9312
C: IBM SP2, PVM, ded.45Mb	68.6±0.1	205±144
D: homogeneous PCs, WinNT, MPI, ded.100Mb	1.04±0.13	6200±7200
homogeneous PCs, Linux, PVM, ded.100Mb	0.833±0.004	1300±400
homogeneous PCs, WinNT, PVM, ded.100Mb	1.59±0.03	24800±3500

may differ from system to system and from implementation to implementation. Thus, they should be understood rather as indicators than the ultimate truth about communication performance. The measurements were taken on unloaded computers (no other user applications were running). The values represent one pair of communicating computers. We do not report results for the Java platform because there is no permission of the software producer.

In Table 3 examples of typical processing rates (A_j) are given. All results refer to a single computer. Note, that these values not only depend on the raw speed of the hardware or the operating system, but also on the application, its implementation, and run-time environment.

Table 3. Examples of processing rates (A_j)

platform	application	A_j[μs/B]
A: various Sun workstations, PVM	search for a pattern	6.99±0.03
B: various PCs, Linux, PVM	compression	1500±20
C: IBM SP2, PVM	compression	650±60
D: homogeneous PCs: WinNT, MPI	search for a pattern	0.838±0.007
D: homogeneous PCs: WinNT, MPI	join	1176±6

Due to space limitations we present only a selection of the results. In the following diagrams difference between the expected execution time and the measurement divided by the expected execution time (i.e. relative error) is presented on the vertical axis. The horizontal axis shows size of the problem. In Fig.2 results of the "search for a pattern" application on platform D are shown. In all cases real running time was longer than the expectation. For platform A the results were similar. The difference is approx. 35% in LIFO case. In the FIFO case the difference has bigger variation, and grows slightly with V from approx.

25% to 30%. In Fig.3 results of the "compression" application on platform C are shown. Real running time was longer than the expectation. LIFO case is more stable and relative error oscillates around 10.5%. In the FIFO case difference is growing with the size of the problem from 6% to 13%. For the same application on platform D relative error decreases from 55% to 7% with increasing V.

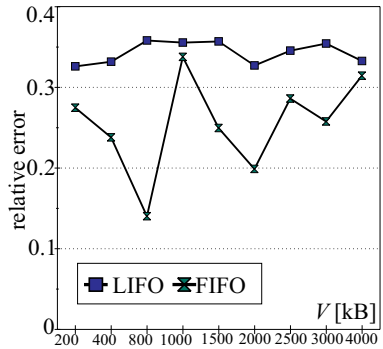


Fig. 2. Difference between model and measurement on platform D in "search for a pattern" application.

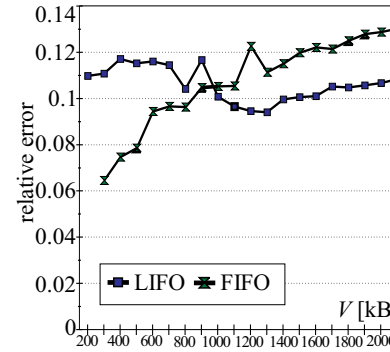


Fig. 3. Difference between model and measurement on platform C in "compression" application.

In Fig.4 relative error for "join" application on platform D is displayed. In both LIFO and FIFO cases the difference decreases from approx. 40% to less than 0.5%. Intuitively, it seems reasonable that there should be a good coincidence between the expectation and the measurement for big values of V, because processing and communication times are long and transient effects are compensated for. In Fig.5 relative difference for "coloring" application on platform F is shown. With growing V the relative error decreases from approx. 30% to less than 5% and then increases to approx. 30%. Real execution time was longer up to 30kB, and from 40kB on it was shorter than the expectation.

5 Discussion and Conclusions

Let us observe that in most of the cases relative difference between the model and the measurement is $\approx$30% and less. We believe that the coincidence of the model and experimental results can be improved if more effort is devoted to better understanding the computing environment, and more carefully setting up the experiments (e.g. if we have more control on the computer software suite). On the other hand, differences below 10% (cf. Fig.3 and Fig.4) indicate that there are applications and platforms where the divisible task model is accurate.

It can be observed that the more uniform and dedicated system we used (e.g. platform C), the better the coincidence with the model was. Calling operating

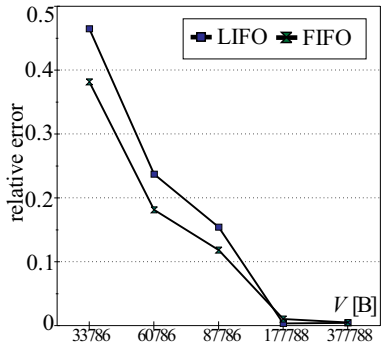


Fig. 4. Difference between model and measurement on platform D in "join" application.

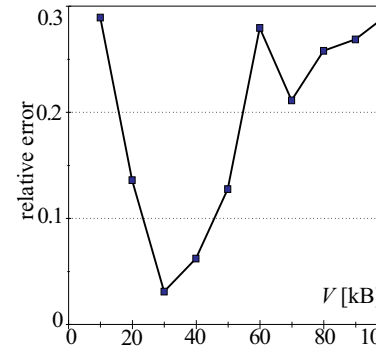


Fig. 5. Difference between model and measurement on platform F in "coloring" application LIFO case.

system and runtime environment services is one of error sources in our results. For example, references to disk files or memory allocation procedures introduces great amount of uncertainty and dependence on the behavior of other software using the computer. This was also the case for long messages for which the efficiency of communication decreased as soon as the message size exceeded free core memory size. Virtual memory was used by the operating system to hold big data volumes to be communicated. In such situations assumption about linear dependence of the communication time on the volume of data was not fulfilled, and communication speed decreased with data size. This observation applies also to the dependence of processing time on the volume of data: in wide ranges of data sizes the assumption on linearity of this function may be not satisfied. Distribution of the results can be another reason for disagreement of the real running time and the expectation. This applies e.g. to "search for a pattern" and "join" applications. In the model, distribution of the results is uniform and any fraction of the total volume of data induces some results. In reality interesting records or text patterns may be abundant in data for one processor, and may be absent from the data for another processor. Our experiments were performed on Ethernet network. The access time to this kind of network is not deterministic. Also the software running in parallel with our programs (e.g. operating system) causes that processing speed is not stable. As a result both communication and computing parameters include some amount of uncertainty, which can be estimated by the value of these parameters standard deviation. The standard deviation of C_j and A_j parameters on most of the platforms was approx. 0.01. Deviation of startup time parameters (S_j) is much bigger, even as much as 3.3 times in the case of platform B.

It has been demonstrated in this work that divisible task model is capable of accurately describing the reality. There are also cases when the predictions of the model are not satisfactory yet. A static and single-chunk distribution of

work was assumed. In everyday practice a dynamic on-line algorithm would be more welcome. Proposing and analyzing such algorithms can be a subject of the future research.

References

1. Bharadwaj, V., Ghose, D., Mani, V., Robertazzi, T.: Scheduling divisible loads in parallel and distributed systems. IEEE Computer Society Press, Los Alamitos CA (1996)
2. Błażewicz, J., Drozdowski, M., Markiewicz, M.: Divisible task scheduling - concept and verification. Parallel Computing **25** (1999) 87–98
3. Cheng Y.-C., Robertazzi, T.G.: Distributed computation with communication delay. IEEE Transactions on Aerospace and Electronic Systems **24** (1988) 700–712
4. Drozdowski, M.: Selected problems of scheduling tasks in multiprocessor computer systems. Poznań University of Technology Press, Series: Rozprawy, No.321, Poznań (1997). Also: `http://www.cs.put.poznan.pl/~maciejd/txt/h.ps`

Optimal Mapping of Pipeline Algorithms[1]

Daniel González, Francisco Almeida, Luz Marina Moreno, Casiano Rodríguez

Dpto. Estadística, I. O. y Computacion, Universidad de La Laguna, La Laguna, Spain
{dgonmor, falmeida, casiano}@ull.es

Abstract. The optimal assignment of computations to processors is a crucial factor determining the effectiveness of a parallel algorithm. We analyze the problem of finding the optimal mapping of a pipeline algorithm on a ring of processors. There are too many variables to consider, the number of virtual processes to be simulated by a physical processor and the size of the packets to be communicated. We provide an analytical model for an optimal approach to these elements. The low errors observed and the simplicity of our proposal makes this mechanism suitable for its introduction in a parallel tool that compute the parameters automatically at running time.

1 Introduction

The implementation of pipeline algorithms on a target architecture is strongly conditioned by the actual assignment of virtual processes to the physical processors, their simulation, the granularity of the architecture, and the instance of the problem to be executed. To preserve the optimality of a pipeline algorithm, a proper combination of these factors must be considered.

The amount of theoretical works [1], [4] contrast with the absence of software tools to solve the problem, most of these solve the case under particular assumptions. Unfortunately, the inclusion of the former methodologies in a software tool is far of being a feasible task.

The *llp* tool presented in [2] allows cyclic and block-cyclic mapping of pipeline algorithms according to the user specifications. We have extended it with a buffering functionality and it is also an objective of this paper to supply a mechanism that allows *llp* to generate automatically the optimal mapping.

[1] The work described in this paper has been partially supported by the Canary Government Research Project PI1999/122.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 320-324, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 The Problem

The pipeline mapping problem is defined as finding the optimal assignment of a virtual pipeline to the actual processors to minimize the execution time. We consider that the code executed by every virtual process of the pipeline is the standard loop of figure 1. The code of figure 1 represents a wide range of situations as is the case of many parallel Dynamic Programming algorithms [2], [3].

The classical technique consist of partitioning the set of processes following a mixed block-cyclic mapping depending on the Grain *G* of processes assigned to each processor. According to the granularity of the architecture and the grain size *G* of the computation, it is convenient to buffer the data communicated into the sender processor before an output is produced. The use of a data buffer of size B reduces the overhead in communications but can introduce delays between processors increasing the startup of the pipeline.

```
void f() {
  Compute(body0);
  While (running) {
    Receive();
    Compute(body1);
    Send();
    Compute(body2);
  }
}
```

Fig. 1. Standard loop on a pipeline algorithm.

We can now formulate the problem: *Which are the optimal values for G and B?*

3 The Analytical Model

Given a parallel machine, we aim to find an analytical model to obtain the optimal values of *G* and *B* for an instance of a problem. The time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes executions has to be modeled. This problem has been previously formulated by [2] using tiling. The size of the tiles must be determined assumed the shape. However, the approach taken assumes that the computational *bodies 0* and *2* in the loop are empty. Also, the considerations about the simulation of the virtual processes are omitted.

When modeling interprocessor communications, it is necessary to differentiate between external communication (involving physical processors) and internal communications (involving virtual processors). For the external communications, we use the standard communication model. At the machine level, the time to transfer *B* words between two processors is given by $\beta + \tau B$, where β is the message startup time and τ represents the per-word transfer time. With the internal communications we assume that per-word transfer time is zero and we have to deal only with the time to access the data. We differentiate between an external reception (β^E) without context switch between processes and an internal communication (β^I) where the context switch must be considered. We will also denote by t_0, t_1, t_{2i} the time to compute respectively *body0*, *body1* and *body2* at iteration *i*.

T_s will denote the startup time between two processors. T_s includes the time needed to produce and communicate a packet of size B. T_c denotes the whole evaluation of G processes, including the time to send M/B packets of size B:

$$T_s = t_0*(G - 1) + t_1 * G * B + G*\Sigma_{i = 1, (B-1)} t_{2i} + 2*\beta^I * (G - 1)* B + \beta^E * B + \beta + \tau *B$$

$$T_c = t_0*(G - 1) + t_1*G*M + G*\Sigma_{i = 1, M} t_{2i} + 2*\beta^I *(G - 1)*M + \beta^E*M + (\beta + \tau*B)* M/B$$

The first three terms accumulate the time of computation, the fourth term is the time of context switch between processes and the last terms include the time to communicate packets of size B.

According to the parameters G and B two situations may appear when executing a pipeline algorithm. After a processor finishes the work in one band it goes to compute the next band. At this point, data from the former processor may be available or not. If data are not available, the processor spends idle time waiting for data. This situation arises when the startup time of processor p (the first processor of the ring in the second band) is larger than the time to evaluate G virtual processors, i. e, when $T_s * p \geq T_c$. Then we denote by R_1 the values (G, B) where $T_s * p \leq T_c$ and R_2 the values (G, B) such that $T_s * p \geq T_c$.

For a problem with N stages on the pipeline (N virtual processors) and a loop of size M (M iterations on the loop), if $1 \leq G \leq N/p$ and $1 \leq B \leq M$ the execution time $T(G, B)$ is:

$$T(G, B) = \begin{cases} T_1(G, B)= T_s * (p - 1) + T_c * N/(G*p) \text{ in } R_1 \\ T_2(G, B) T_s * (N/G - 1)+T_c \text{ in } R_2 \end{cases}$$

Fixed the number of processors p, the parameters β^I,β^E, β and τ are constants architectural dependent and t_0, t_1, t_{2i} , M and N are variables depending on the instance of the problem. The actual values for these variables are known at running time. An analytical expression for the values (G, B) leading to the minimum, will depend on the five variables and seems to be a very complicated problem to solve. Instead of an analytical approach we will approximate the values for (G, B) numerically. An important observation is that $T(G, B)$ first decreases and then increases if we keep G or B fixed and move along the other parameter. Since, for practical purposes, all we need is to give values for (G, B) leading us to the valley of the surface, a few numerical evaluations of the function $T(G, B)$ will be sufficient.

To introduce the model into a tool that automatically computes G and B, during the execution of the first band, the tool estimates the parameters defining the function $T(G, B)$ and carries out the evaluation of the optimal values of G and B. The overhead introduced is negligible, since only a few evaluations of the objective function are required. After this first test band, the execution of the parallel algorithm continues with the following bands making use of the optimal Grain and Buffer parameters.

4 Validation of the Model

We have applied the model to estimate the optimal grain *G* and optimal buffer *B* for the 0-1 Knapsack Problem (KP) [3] and the Resource Allocation Problem (RAP) [2]. A pipeline algorithm for the RAP has the property that *body2* does not take constant time. Table 1 presents the values for *(G-Model, B-Model)* obtained with the model, the *llp*-running time of the parallel algorithm for this parameters (*Real Time*), the running times obtained with the best values of *(G-Real, B-Real)* and the best running time (*Best Real Time*). The table also shows the error made ((*Best Real Time - Real Time*) / *Best Real Time*) when we consider the parameters provided by the tool instead of the optimal values.

The model shows an acceptable prediction in both examples with an error not greater than 15 %.

Table 1. Estimation of *G*, *B* for the KP and RAP.

	P	G-Model	B-Model	Real Time	G-Real	B-Real	Best Real Time	Error
KP	2	10	2048	140.08	20	5120	138.24	0.003
KP	4	10	768	70.84	20	1792	69.47	0.053
KP	8	10	512	35.85	20	768	35.08	0.097
KP	16	10	192	18.29	10	768	17.69	0.150
RAP	2	10	10	73.33	5	480	70.87	0.034
RAP	4	5	10	36.73	5	160	36.01	0.020
RAP	8	2	10	19.26	5	40	18.45	0.044
RAP	16	1	10	10.79	5	40	9.57	0.127

5 Conclussions

We have developed an analytical model that predicts the effects of the Grain of processes and Buffering of messages when mapping pipeline algorithms. The model allows an easy estimation of the parameters through a simple numerical approximation. The model is capable to be introduced into tools (like *llp*) that produce the optimal values for the Grain and Buffer automatically.

References

1. Andonov R., Rajopadhye S.. Optimal Orthogonal Tiling of 2D Iterations. Journal of Parallel and Distributed computing, 45 (2), (1997) 159-165.
2. Morales D., Almeida F., Garca F., Gonz lez J., Roda J., Rodrguez C.. A Skeleton for Parallel Dynamic Programming. Euro-Par 99 Parallel Processing Lecture Notes in Computer Science, Vol. 1685. Springer-Verlag, (1999) 877 887.

3. Morales D., Roda J., Almeida F., Rodr guez C., Garc a F.. Integral Knapsack Problems: Parallel Algorithms and their Implementations on Distributed Systems. Proceedings of the 1995 International Conference on Supercomputing. ACM Press. (1995) 218-226.
4. Ramanujam J., Sadayappan.. Tiling Multidimensional Iterations Spaces for Non Shared-Memory Machines. Supercomputing 91. (1991) 111-120.

Dynamic Load Balancing for Parallel Adaptive Multigrid Solvers with Algorithmic Skeletons

Thomas Richert

Lehrstuhl f. Informatik II, RWTH Aachen, 52056 Aachen, Germany,
rich@i2.informatik.rwth-aachen.de

Abstract. Algorithmic skeletons are polymorphic higher-order functions that represent common parallelization patterns. In this paper we present a parallel implementation of a skeleton-based dynamic load balancing algorithm for parallel adaptive multigrid solvers. It works on distributed refinement trees that arise during adaptive refinement of grids. Finally, we discuss some properties of the algorithm, for example speed and locality of the distribution.

1 Introduction

Adaptive multigrid algorithms are the best known methods for solving partial differential equations numerically on a sequential computer [2]. Parallelization of these algorithms means to extend it, so that they work with distributed grids. After adaptive refinement the distribution of the grid elements over the processors is often in imbalance. Hence, we have to implement a *dynamic load balancing algorithm* (DLBA) that moves some nodes and elements from one processor to another.

Unfortunately, the implementation of an adaptive multigrid algorithm on parallel computers is a difficult and erroneous task, because often parallel programmers have to rely on low-level message passing functions. Our approach to facilitate parallel programming is based on *algorithmic skeletons* [3].

In this paper we describe the parallel implementation of a skeleton-based DLBA for parallel adaptive multigrid algorithms that works on refinement trees [4]. A *refinement tree* arises with adaptive refinement and records the way of refinement. Because of the distribution of the grids the refinement tree is distributed, too. Overlapping of distributed grids implies that parts of the distributed tree occur on more than one processor. Especially the root of the tree is stored on every processor. We establish connections among parts of the tree by tagging some nodes as *virtual leafs* and *link nodes*. Virtual leafs are nodes with subtrees that are stored at link nodes on other processors.

2 Algorithmic Skeletons with *Skil*

A *skeleton* is an algorithmic abstraction common to a series of applications that can be implemented in parallel. Skeletons are embedded in a sequential

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 325–328, 2000.
© Springer-Verlag Berlin Heidelberg 2000

host language, thus being the only source of parallelism in programs. The basic idea of algorithmic skeletons relies on the paradigm of functional languages: based on techniques like *higher order functions*, *type polymorphism*, and *partial application* we can write flexible and reusable skeletons that can be instantiated for each application individually. In this paper, we use *Skil* (*Sk*eleton *I*mperative *L*anguage) [1] to implement our skeletons. To avoid the lack of efficiency in pure functional programs *Skil* is an extension of C. The *Skil* compiler translates code from *Skil* into C by instantiating the skeletons with application-dependent types and functions.

For the implementation of the DLBA we need a `fold`-like skeleton that works on distributed trees. Because of the topology of distributed trees we have to define a new parallel algorithm to implement the skeleton `fold_t`. If the tree is distributed over more than one processor, `fold_t` performs the following steps. First, for each local part of the tree the data of all elements are combined from leafs to root in parallel. Then the data at virtual leafs have to be updated by getting data from processors that hold respective link nodes with a real subtree. At last the received data have to be mapped to all nodes that are above the virtual leafs in the tree. The result is stored at the root of the whole tree. The *Skil* prototype declaration of this skeleton is given by

```
$u fold_t(Tree <$t> tree, $u get_f($t), $t store_f($u), $u fold_f($u, $u));
```

The first argument of `fold_t` represents the distributed tree. Each node of the tree has the polymorphic type `$t` that must be instantiated by the user of this skeleton. The other arguments are variables that stand for user-defined functions. The type variable `$u` have to be instantiated with the computation type, for example `float`. With `get_f` we get the stored data of type `$u` from the tree node of type `$t`. The function `store_f` stores data into a suited entry of the tree node. Moreover, for combining values of type `$u` the user of `fold_t` has to define the binary operation `fold_f`. For instance, possible operations are binary addition or the maximum function.

Additionally, for dynamic load balancing we need a mechanism to transfer parts of a grid from one processor to another. In [5] we describe the skeletons that we designed and implemented for this purpose. Especially, we presented the skeletons `copy` and `delete` to move grid objects between processors. To avoid communication overhead these operation are not executed immediately. We collect all necessary data and store it into a *communication table*. After that we use this table and the skeleton `execute_transfer` to perform communication in one step. In [5] we analyzed the object transfer skeletons and showed the efficiency of our implementation.

3 Dynamic Load Balancing with Skeletons

In general, a DLBA has the following structure:

1. compute current distribution of load
2. use a strategy for computing necessary actions for load balancing
3. perform communication to do load balancing

Note that communication is only necessary in the first and the last step. We implement the first phase by calling the `fold_t` skeleton:

```
ntris = fold_t(refinement_tree, get_weight, store_weight, (+));
```

After execution the root node of each part of the distributed tree contains the number of all triangles `ntris` in the finest multigrid level. The second phase starts with computing of the number of triangles per partition `m` by dividing `ntris` by the number of processors `p`. Furthermore, the algorithm needs an array of integers `count[]` that is used for checking available space in the partitions. The k-th entry of `count[]` represents the current partition size of the k-th processor. Note that each processor `q` holds it own `count[]` array. The `q`-th entry of the array is set to `m`. The initialization of the other entries depends on the weight information on the respective virtual leafs. If a processor holds more or equal to `m` triangles, the respective entry in `count[]` is set to zero. Otherwise the entry contains the size of available space in the partition. The next step consists of calling the recursive function `rt_balance`:

```
proc rt_balance(NodeOfTree node, int count[], int m)
    q = get_current_index(node);
    children = compute_children_order(node);
    for all children of node
        if (weight(child) > count[q])
            rt_balance(child, count[], m);
        else
            decrement count[i] by weight(child);
            if (q != myProc)
                copy(triangle(child), q, tri_dep_f);
                delete(triangle(child));
            endif
        endif
    endfor
endproc
```

The index computation assures that as much as possible of the current part of the distributed tree should remain on the processor to avoid unnecessary movement of grid objects. The determination of a order in which the children of a node will be traversed is necessary to assure locality of the distribution on a grid level. However, this depends on the element shape and refinement technique. If the subtree with root `child` does not fit into the current partition, the algorithm must go deeper in this subtree. Otherwise, the subtree is added to the current partition by decrementing the respective counter `count[q]`. If `q` is not the number of the processor where the algorithm runs, it has to call the operations `copy` and `delete` of the object migration mechanism. The dependency function `tri_dep_f` provides copying and removing of all triangles and grid nodes that occurs in the subtree. The last phase consists of performing communication by calling `execute_transfer` and updating the refinement tree.

4 Properties

Let N be the number of triangles, p be the number of processors, and M the number of transfer objects. The asymptotic time complexity of the algorithm is $O(N)$, because the first phase takes $O(N/p)$ operation, the second $O(p \log N)$ operations, and the last M operations. Note that $M << N$ and `copy` and `delete` can be executed in constant time. Because N is much larger than M and in parallel multigrid solvers should be N much larger than p, communication time is neglectable here.

Moreover, the partitioning algorithm produces an optimal balance [4]. It is not difficult to show that the algorithm provides locality of the distribution both on each grid level and among different levels. However, the algorithm does not minimize the number of edges that are cut by the partition. This disadvantage is not very important because the number of multigrid cycles inside the loop of the adaptive multigrid algorithm is low. Moreover, if we use a suitable overlapping strategy, we do not need more than two communication phases per multigrid cycle [6].

5 Conclusions and Future Work

We presented a skeleton-based parallel implementation of a DLBA working on a distributed refinement tree that arises from adaptive grid refinement. The results we have obtained support the idea that the use of skeletons to hide communication leads to efficient programs, which are smaller and easier to understand than comparable low-level implementations.

The next step is to integrate this DLBA into a multigrid solver with adaptive grid refinement using the presented skeletons. We want to investigate if such a project can be implemented in *Skil* with comparable efficiency but with less programming effort compared to a low-level implementation.

References

1. Botorog, G.H, Kuchen, H: *Skil*: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In: Proceedings of HPDC-5 '96, IEEE Computer Society Press (1996) 243-252
2. Brandt, A.: Multi-level Adaptive Solutions To Boundary-Value Problems. Math. of Comp. 31 (1977) 333-390
3. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman/MIT Press, London (1989)
4. Mitchell, W.: The Refinement-Tree Partition for Parallel Solution of Partial Differential Equations. NIST Journal of Research 103 (1998) 405-414
5. Richert, T.: Management of Dynamic Distributed Data with Algorithmic Skeletons. Parallel Computing - Fundamentals and Applications - Proceedings of the International Conference ParCo99 (1999) To appear.
6. Richert, T: Using Skeletons to Implement a Parallel Multigrid Method with Overlapping Distributed Grid. In: Proceedings of PDPTA'2000. Las Vegas, USA (2000) To appear.

Topic 04
Compilers for High Performance

Samuel P. Midkiff, Barbara Chapman, Jean-François Collard, and Jens Knoop

Topic Chairpersons

We would like to welcome you to the Euro-Par 2000 topic on High Performance Compilers. The presentations, papers, and interactions with fellow researchers promise to be both enjoyable and useful. We also hope that you enjoy your visit to Munich.

The High Performance Compilers topic is devoted to research in the areas of static program analysis and transformation, mapping programs to processors (including scheduling, allocation and mapping of tasks), code generation and compiling for heterogeneous systems. The topic is distinguished from the other compiler oriented topics — #03, #07 and #14 — in Euro-Par by its focus on the application of these techniques to the automatic extraction and exploitation of parallelism.

This year 27 papers, from three continents and seven countries, were submitted. The quality and range of the submitted papers was impressive, and testifies to the continued vibrancy and importance of the field of compilation for high performance computing. All papers were reviewed by three or more referees. Using the referees' reports as guidelines, the program committee picked eleven of the submitted papers for publication and presentation at the conference. Ten papers were selected as regular papers, and one as a research note. These will be presented in four sessions, one of which is a combined session with topic 14, Instruction Level Parallelism and Processor Architecture.

The paper in the combined session with topic 14 (Session 14-B.1 on Thursday afternoon), by Kevin D. Rich and Matthew K. Farrens, describes an implementation of a compiler that automatically partitions code between data read/write operations and data manipulation operations to target processors that support independent data fetch and write instruction streams. This allows more flexibility and parallelism in the scheduling of memory references and computation.

The first full session of the High Performance Compilers topic focuses on automatic parallelization. Reflecting the increasing importance of sparse computations in both industrial and basic science applications, the first two papers describe methods for improving compiler parallelization of sparse code. The first of these, by Gerardo Bandera and Emilio L. Zapata, uses information from high-level directives to perform sparse privatization and thereby parallelize the sparse code. The second paper, by Roxane Adle, Marc Aiguier and Franck Delaplace, uses symbolic analysis to perform more precise dependence analysis on sparse structures to parallelize the sparse codes. The third paper, by Rashindra Manniesing, Ireneusz Karkowski and Henk Corporaal, targets problems at the equally

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 329–330, 2000.
© Springer-Verlag Berlin Heidelberg 2000

important, but opposite end of the computation spectrum — SIMD parallelization of embedded applications using a pattern matching based code generation phase.

The first three papers of the second full session are concerned with program restructuring. The first paper describes the management of temporary arrays arising from the distribution of loops with control dependences. In particular, Alain Darte and Georges-André Silber describe a new type of dependence graph and how it is used to limit the number of temporaries used to store conditionals. The second paper, by Nawaaz Ahmed and Keshav Pingali, describes how block-recursive codes can be automatically generated from iterative versions of a kernel to better exploit data locality across the entire memory hierarchy. The third paper, by Nikolay Mateev, Vijay Menon and Keshav Pingali, describes how to transform linear algebra computations between eager (or right-looking forms) and lazy (or left-looking forms) to enhance later compiler optimizations. Finally, the last paper of the session, by Diego Novillo, Ron Unrau and Jonathan Schaeffer, examines the problem of validating irregular mutual exclusion synchronization in explicitly parallel programs.

The third, and final full session focuses on problems of data layout and parallelism specification. In the first paper, R. W. Ford, M. F. P. O'Boyle and E. A. Stohr show how to place a minimal number of coherence operations in such a way as to eliminate all invalidation traffic in programs with statically decidable control flow. The second paper, by Alain Darte, Claude Diderich, Marc Gengler and Frédéric Vivien, presents a technique to consider together the mapping of loop iterations to processors, and the order of execution (schedule) of those iterations, for better exploitation of parallelism than can be achieved by using strategies which independently arrive at mappings and schedules. Finally, the third paper, by Felix Heine and Adrian Slowik, describes how to use Ehrhart polynomials to precisely determine the amount of static locality, and to use this information to guide data transformations and distributions to increase the quality of a program's data distribution.

In closing, we would like to thank the authors who submitted a contribution, as well as the Euro-Par Organizing Committee, and the scores of referees, whose efforts have made this conference, and the High Performance Compilers track possible.

Improving the Sparse Parallelization Using Semantical Information at Compile-Time*

Gerardo Bandera and Emilio L. Zapata

Department of Computer Architecture, University of Málaga,
P.O. Box 4114, E-29080 Málaga, Spain
{bandera,ezapata}@ac.uma.es

Abstract. This work presents a novel strategy for the parallelization of applications containing sparse references. Our approach is a first step to converge from the data-parallel to the automatic parallelization by taking into account the semantical relationship of vectors composing a higher-level data structure. Applying a *sparse privatization* and a *multi-loops analysis* at compile-time we enhance the performance and reduce the number of extra code annotations. The building/updating of a sparse matrix at run-time is also studied in this paper, solving the problem of using pointers and some levels of indirections on the left hand side. The evaluation of the strategy has been performed on a Cray T3E with the matrix transposition algorithm, using different temporary buffers for the sparse communication.

1 Introduction

Research about irregular computation is presently taking more importance, though most of the parallelization techniques are only focused on dense operations. Real scientific algorithms spend the major part of its execution time in sparse matrix computations. They increment the complexity of the parallelization due to the presence of indirections. In the other hand, new algorithms contain high-level data structures, composed by several vectors. Though current compilation techniques handle all these components individually, an efficient parallelization necessitates a different approach. Hence, our first goal in this paper is to demonstrate that the performance of the SPMD code is enhanced if the **semantical relationship** between the data-structure components is considered.

During the last years, some works about the sparse parallelization have been developed [3,7,5]. All of these approaches intend to improve the performance by a special analysis and transformation of this part of the code. From our point of view, none of them are very efficient with real sparse algorithms, because they do not use semantic information at compile-time. Additionally, these methods are keeping away from an automatic parallelization by requiring more information from users during the compilation.

* The work described in this paper was supported by the Ministry of Education and Culture (CICYT) of Spain under project TIC96-1125-C03.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 331–339, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In our previous works we have demonstrated the utility of the *SPARSE* directive to define a sparse data-structure [10,1,2]. With this annotation we remark the presence of *semantical bindings* between vectors composing the matrix, caused by the affinity of their information. To complement it, the *DISTRIBUTE* directive must be also included, to divide matrix entries onto the processors by means of a sparse block-cyclic distribution. The use of a pseudo-regular distribution instead of the traditional regular produces the same storage format for every local matrix than the representation of the global one.

This work describes a new feature in our compilation support: the parallelization of a run-time sparse matrix building/updating algorithm. It has a remarkable importance because compressed representations typically imply the use of *pointers* in code instead of coordinates. Although pointers analysis [6] grows in importance for recent applications, there are not many works addressing this problem. Our solution is based on the privatization [9], an important technique used in automatic parallelizers. The compilation strategy presented here is mainly focused on the data-parallel programming model by extending the meaning of some HPF directives. Nevertheless, we attempt to converge to the automatic parallelization of algorithms including complex data-structures by using contiguous loops analysis and pointers-to-coordinates translations. To complete our analysis, we also include a temporary storage study. It is required to optimize the performance of the sparse information communication. We will present three buffer prototypes which will be tested within the sparse transposition algorithm parallelization on a Cray T3E. The rest of the paper is divided in the following sections: Section 2 describes the compilation support for sparse readings and writings, and a sending buffer analysis for applications containing sparse communications. Section 3 includes the parallelization of an interesting case of study by using our compile-time scheme. The evaluation of our proposal will be displayed in section 4, and finally, the conclusions.

2 Compilation Strategy Based on Privatizations

This section describes the compile-time analysis used on the parallelization of applications containing sparse references. We present here the loops partitioning for matrix readings, its extension for sparse writings and a temporary storage selection for algorithms including sparse information interchange.

2.1 Sparse Loops Partitioning

Figure 1.a depicts the typical pair of nested loops accessing the non-null entries of a compressed by rows (CRS) matrix. Vector *DA* stores the entries values and *RO* the row pointers. *X* is a dense vector which is being updated. Applying the *owner computes rule* we will obtain the SPMD code showed in figure 1.b. Two additional pre-processing stages are required: for the calculation of the local bounds of the inner loop "j", and for the non-local entries accessed. This alternative is the

```
DO i = 1, n
  DO j =  RO(i),  RO(i+1)-1
    X(i) = .....  DA(j) .....
  ENDDO
ENDDO
```

(a)

```
DA vector Pre-Processing →  newDA
RO vector Pre-Processing →  newRO

DO i = local-iteration
  DO j = local-iteration with  newRO
    X(i) = ...  newDA ...
  ENDDO
ENDDO
```

(b)

```
DO i = local-iteration
  DO j =  RO(i),  RO(i+1)
    X(i) = ...  DA(j) ...
  ENDDO
ENDDO

X Vector Post-Processing
```

(c)

Fig. 1. *(a) Sequential code reading the sparse matrix entries and updating a dense vector X; (b) Parallel code applying the owner computes rule to (a); (c) Parallel code after a sparse privatization: DA and RO vectors contain now local information.*

typical *inspector/executor*, where two additional vectors (*newRO* and *newDA*) are filled after the loop execution by costly stages.

The local representation on each processor, caused by a pseudo-regular distribution, recommends an alternative parallelization taking into account the semantical relationship between *DA* and *RO* vectors. In this way, figure 1.c depicts the new parallel version of the sequential code showed in figure 1.a, where no pre-processing stages are required. This strategy, called **sparse privatization**, consists in the following: Every processor will only compute iterations involving its local sparse data, making private copies of remote data accessed in statements RHS, if necessary. Finally, if no locality is found, the compiler will include a post-processing stage to broadcast private results to the owner processors. With this solution we avoid costly sparse communications, trying to compute as many local operations as possible, and replacing *Gather* operations by *Scatters*.

2.2 Sparse Matrix Updating

Many real applications not only contain loops with sparse readings, but also includes some matrix writings (i.e, sparse addition, multiplication, transposition, LU decomposition,...). Existing parallelizations of these algorithms have two main problems: achieve a poor performance and uses complex data structures [1]. Our aim now is to extend the sparse privatization previously presented to the matrix updating in order to obtain an efficiency as close as possible to the hand-made parallel code.

Typically, the code including a sparse matrix updating is composed by some loops writing the different vectors of the matrix. Most of the related work about loops parallelization are focused to single loop partitions. However, this solution produces a poor performance with sparse updating algorithms. Therefore, our alternative analyses contiguous loops of the sequential code to detect writings on the different vectors of the matrix. After the first modification detection, the compiler will continue analyzing the remaining code to carry out a combined transformation based on the semantic information of the data-structure. It only implies the analysis of a reduced number of branches of the abstract syntax tree (2 or 3 at most), but the performance will be very much improved.

As pseudo-regular distributions produce a different partition of the matrix composing vectors, it also implies a different compile-time translation of every sparse loop: (1) Pointers vector writing will produce private writings on every processor; (2) Compressed vectors updatings are also locally computed in a first step. The local pointers vectors will be used for the placement of the compressed vectors information on each processor. This compressed information will be placed on the corresponding processors in a second step.

A simplification of the parallelization process is depicted in figure 2. Note that this transformation will be completed when the data vector (DA) modification had been detected. This is not a mandatory, but an important percentage of real codes follow this requirement. Nevertheless, a generic parallelization process is also included in our compilation support, with a more costly semantical dependence analysis. As readers can observe, the parallelization of loops containing a coordinates vector modification (RO and CO) is done by creating private copies. This local information must be stored on temporary buffers, which will be sent afterwards to the appropriate destinations. The DA updating detection implies the inclusion of two additional routines within the SPMD code: a Communication and a Matrix Reconstruction. If the data modification is not detected in the following loops, the parallelization will be completed only with new coordinates. The number of loops to analyze after the CO and RO updating will depend on an input parameter of the parallelization tool, which will be fixed at compile-time.

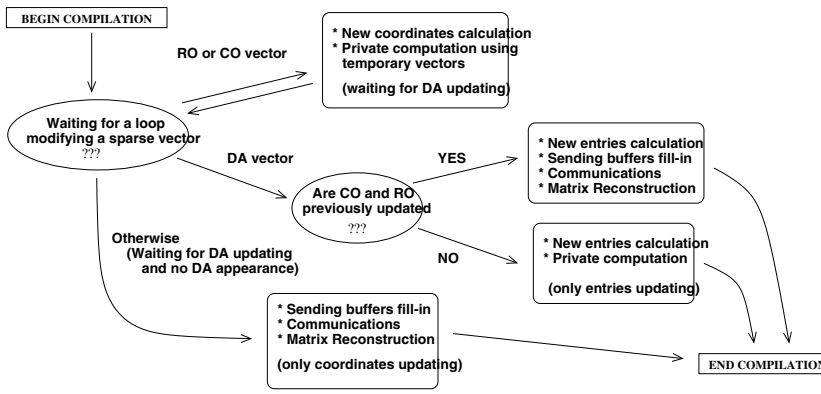


Fig. 2. *Compiler strategy for algorithms containing a sparse matrix updating.*

2.3 Buffering Analysis

A temporary storage study is necessary when the application to parallelize requires sparse data interchange between processors. The selection of an efficient data-structure to allocate the information involved in the communication has

a remarkable influence in the parallel performance: first, in the Collecting and Mixing stages, with a different index processing; and second, in the Communication time, with the necessity of an implicit coordinates storage. Hence, we have implemented the three following alternatives for the sparse data interchange: the **Unsorted Buffer**, the **Linked-Lists** and the **Histogram Buffer**.

In the *Unsorted Buffer*, source processors pack the matrix entries in the same order they are visited. As it does not typically coincide with the order in destination, an explicit inclusion of coordinates will be required for the matrix reconstruction. The memory occupation of this buffer only depends on the maximum number of elements to send to a single processor. As this value is not known at compile-time, a good estimation is required to avoid a memory overhead. *Linked-Lists* are based on dynamic memory allocations and pointers arrangements. Every data entry is stored in a cell with one of the coordinates, while the second one is used to select the list where this cell will be linked to. The number of cells on a given list indicates the non-nulls of this row. In this buffer the cell allocation is done by demand, so the memory reservation will be minimal and no estimation is required. The *Histogram Buffer* is also composed by three vectors. The first two store in a sorted fashion data entries and one coordinate, while a third vector contains counters of elements belonging to the same dimension. While the length of this last vector coincides with the number of rows in destination, the first two are divided in slices of the same size, where the elements belonging to the same row are placed. A careful slices size estimation is needed, because the different occupation percentage of every row can produce many memory wasting.

3 Parallelization of the Matrix Transposition

This section describes the parallelization of the sparse transposition algorithm using our compile-time strategy. The selected code is a very motivating example containing three level of indirections in some statement LHS. The data-parallel version of the transposition is based on the sequential code developed by Pissanetzky in [8]. This is the most efficient sequential algorithm, in spite of its strange code structure. There exists a second alternative for this algorithm, which is simpler to understand but performs worse. The HPF code is shown in figure 3. We declare a $N \times M$ CRS sparse matrix A with vectors *(DA, CO and RO)* and *alpha* non-null entries. The transposed matrix *newA* is also defined in a similar way using another triplet of vectors. With the *DISTRIBUTE* and the first *ALIGN* directive, we are specifying a SPARSE-CYCLIC(k) distribution for both matrices. The second alignment is for the dense vector $ROW2$. It is used as an extra pointers vector to avoid costly memory occupations and an additional classification step. We have extended the meaning of this directive. When a dense vector is aligned with a pointers vector of a sparse matrix, we specify the same distribution for both the alignee and the target of the alignment. Thus, they will only have a local meaning.

```
!HPF$ PROCESSORS, DIMENSION(NPES) :: linear
      REAL, DIMENSION(alpha):: DA, newDA
      INTEGER, DIMENSION(alpha):: CO, newCO
      INTEGER, DIMENSION(N+1):: RO
      INTEGER, DIMENSION(M+1):: newRO
      INTEGER, DIMENSION(M+2):: ROW2
!HPF$ REAL,DYNAMIC,SPARSE(CRS(DA,CO,RO))
!HPF$+ :: A(N,M)
!HPF$ REAL,DYNAMIC,SPARSE(CRS(newDA,newCO,
!HPF$+ newRO)) :: newA(M,N)

!HPF$ DISTRIBUTE (CYCLIC(k),*) ONTO linear :: A
!HPF$ ALIGN newA(I,J) WITH A(I,J)
!HPF$ ALIGN ROW2(I) WITH newRO(I)
      ...
      ! Reading Matrix A from file
      ...
      ROW2(1:M+2) = 0

!HPF$ INDEPENDENT
      DO 10 I= 1, N
!HPF$ ON HOME(RO(I)), RESIDENT()
        DO 20 J= RO(I), RO(I+1)-1
          ROW2(CO(J)+2) = ROW2(CO(J)+2) + 1
20      ENDDO
10    ENDDO

!HPF$ ON HOME(ROW2(*)) BEGIN
      ROW2(1) = 1
      ROW2(2) = 1
      newRO(1) = 1
      DO 30 I= 3, M+1
        ROW2(I) = ROW2(I) + ROW2(I-1)
        newRO(I-1) = ROW2
30    ENDDO
!HPF$ END ON

!HPF$ INDEPENDENT
      DO 40 I= 1, N
!HPF$ ON HOME(RO(I)), RESIDENT(ROW2)
!HPF$+ BEGIN
        DO 50 J= RO(I), RO(I+1)-1
          newCO(ROW2(CO(J)+1))) = I
          newDA(ROW2(CO(J)+1)) = DA(J)
          ROW2(CO(J)+1))++
50      ENDDO
!HPF$ END ON
40    ENDDO
```

Fig. 3. *HPF Sparse Matrix Transposition.*

The data-parallel code can be decomposed in two main parts: the pointers vector of the new matrix (*newRO*) is calculated in the first part of the code, while *newDA* and *newCO* are filled in the second. The first statement after the file reading is parallel in fact, because it is written using Fortran90. The partition of loop 10-20 is driven by the *INDEPENDENT* and *ON HOME* annotations. They indicate a parallel execution on each processor using their local submatrices, and also obtaining private results. In the next part of the code, together with the *ON HOME* directive, writings on pointers vectors also indicates this privacy. Thereby, every processor will use and calculate private values of vectors *ROW2* and *newRO*.

The last part of the algorithm is the data and column vectors updating (loops 40-50). At this moment of the compilation, the pointers vector of the new matrix has being already modified. In the same way that loops 10-20, the *INDEPENDENT* and *ON HOME* directives cause a sparse privatization, where every processor will compute a set of iterations with its local submatrices. As we have depicted in figure 2, the *newDA* writing causes the completion of the matrix updating. Hence, the compiler will include a *Collecting* stage in order to fill the sending buffers in. Moreover, a *Communication* stage and a final matrix reconstruction (*Mixing*) will be included after the loop execution. For this last part of the parallelization, the compiler must select one of the three buffering alternatives presented in section 2.3.

By analyzing the different parts of the HPF code, we have deduced that some annotations can be removed. The main requirements of our compilation support are the declaration part annotations: the SPARSE directive, because it defines the semantic relationship between the different vectors composing the matrix; and the DISTRIBUTE and ALIGN directives, specifying the owner processors of every matrix entry. Two main details of this concrete application make sure its automatic parallelization: (1) *The loops bounds*: from the above directives the

compiler knows that loops 10-20 and 40-50 are used to visit the matrix A by rows, and thus, every processor will execute different loops iterations only with its local submatrices (sparse privatization); (2) *The LHS vectors*: while pointers vectors writings imply private computation, data vector updating require the completion of the transformation, including Collecting, Communication and Mixing stages.

4 Experimental Results

In this section we evaluate the efficiency of our compilation with the case of study presented in section 3. We have used the Cray T3E, SHMEM routines and the *cc* compiler with the $-O2$ turned on. We have tested different matrices and distribution parameters, but we only include here results for two large matrices from the Harwell-Boeing Collection: a very sparse matrix (BCSSTK30 or **B30**) containing 1036208 non-nulls and with 28924 rows and columns (density rate = 0.12%), and a very dense sparse one (PSMIGR1 or **PS1**), with order 3140 and 543162 entries (5.51%).

The first evaluation is about the influence of the sending buffer in the performance of the transposition. Figure 4 shows the total time of the algorithm for the three buffering schemes previously described. As we can observe, the best performance version uses the *Histogram Buffer*, because of the nice cache behavior of the sorted information. Although the memory occupation is lesser, the necessity of sorting the entries at destination produces an important delay when using the *Unsorted Buffer*. Finally, the worst buffer selection is the *Linked Lists*, where the code overhead is incremented with the idle time produced by continuous cells allocations. Nevertheless, this alternative is the only one useful with very large matrices. Buffers enhancements are more remarked for dense matrices, because the number elements to store on every dimension grows.

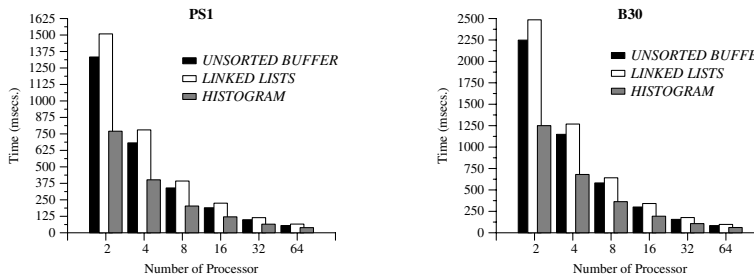


Fig. 4. *Execution time with every buffering alternative. PS1 and B30 matrices.*

A different way of testing the powerful of our compilation strategy is performing a comparison with the typical run-time support using pre and post-processing stages for code indirections. Previous works [10,1] have illustrated the benefits

of similar approaches to ours in comparison with CHAOS[11] for matrix readings. For sparse writings, the resulting code with CHAOS increments the delay, because it will need many more pre-processing stages. The expected results with PILAR[4] are very similar, because even improving the CHAOS performance, the sparse relationship between the dense vectors composing the matrix has not been taken into account. For the same reason, our approach also increments the performance regarding to traditional sparse solvers. The excellent scalability of the translated code must be underlined, in spite of transposition mainly performs data movements. In the same way, we have also obtained an efficient parallelization, because the time of the sequential version (370.71 msec. for PS1 and 231.59 msec. for B30) is improved from 8 and 16 processors, respectively.

5 Conclusions

Sparse references increment the complexity of the parallelization, due to the presence of many code indirections and a replacement of coordinates by pointer values. In the other hand, many sparse applications present computation locality, which can be exploited by providing information to the compiler about the structure and the data distribution. This information is enough to improve the parallel performance. The parallelization support presented in this work is based on the semantical relationship of the different vectors composing a high-level data-structure. It is denoted by the SPARSE directive. This one, jointly with the use of a pseudo-regular distribution, implies the replacement of the owner computes rule by a *sparse privatization* approach, where the computing processor is the sparse entries owner. At the same time, a multi-loop analysis is also enabled. With our solution, costly pre-processing stages and sparse communications are removed.

The dynamic sparse building/updating has also been addressed in this paper. Our compilation algorithm has been described and tested with a remarked application: the transposition. Parallel codes containing sparse communications also require a buffering study. We have presented here three alternatives for storing data entries and coordinates. They will be useful depending on the memory limitations. Although the parallelization approach presented here is based on a sequential code annotation, it constitutes a first step to the automatic parallelization of applications containing high-level data structures.

References

1. R. Asenjo. *LU Sparse Matrices Factorization on Multiprocessors.* PhD thesis, Computer Architecture Dept., University of Málaga, 1997.
2. G. Bandera. *Semi-Automatic Parallelization of Applications containing Sparse Matrices.* PhD thesis, Computer Architecture Dept, University of Málaga, 1999.
3. A.J.C. Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, University of Leiden, The Netherlands, 1996.
4. D.R. Chakrabarti, N. Shenoy, A. Choudhary, and P. Banerjee. An efficient uniform run-time scheme for mixed regular-irregular applications. In *Proc. of ICS'98.*

5. F. Delaplace and R. Adle. Extension of the dependence analysis for sparse computation. In *Proc. of Parallel and Distributed Computing Systems*, October 1997.
6. R. Ghiya and L.J. Hendren. Putting pointer analysis to work. In *Proc. of the* 25th *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 1998.
7. V. Kotlyar, K. Pingali, and P. Stodghill. Compiling parallel code for sparse matrix applications. In *Proc. of Supercomputing*, 1997.
8. S. Pissanetzky. *Sparse Matrix Technology.* Academic Press Inc., 1984.
9. P. Tu and David Padua. Automatic array privatization. *Sixth Workshop on Languages and Compilers for Parallel Computing*, 1993.
10. M. Ujaldón, E.L. Zapata, B. Chapman, and H.P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Trans. on Parallel and Distributed Systems*, 8(10):1068–1083, 1997.
11. J. Wu, R. Das, J. Saltz, and H. Berryman. Distributed memory compiler design for sparse problems. *IEEE Trans. on Computers*, 44(6):737–753, June 1995.

Automatic Parallelization of Sparse Matrix Computations: A Static Analysis

Roxane Adle, Marc Aiguier, and Franck Delaplace

Université d'Évry Val d'Essonne, CNRS EP738, LaMI. F-91025 Évry Cedex, France
fax number: 33 (+1) 69 47 74 72
{adle,aiguier,delapla}@lami.univ-evry.fr

Abstract. This article deals with the definition of a new method for the automatic parallelisation of sequential programs working on dense matrices for generating a parallel counterpart working on sparse matrices.

Keywords. fill-in, non-standard semantics, sparse dependence analysis, Bernstein's conditions.

Introduction

Numerical applications using sparse matrices are ubiquitous in science and engineering such as dynamic fluids or mechanical structure computations. Parallel programs dealing with sparse matrices are considered to be error-prone, hard to conceive and difficult to support. Thus, it is important to develop restructuring compilers to automatically transform numerical programs into equivalent ones making sparse matrix computations. In this way, two works have mainly been proposed [3,6]. In [3], the authors have based their compiler MT1 by using data structures as CRS (Compressed Row Storage) or CCS (Compressed Column Storage) for storing sparse matrices. Program transformations are formalized using polyhedral algebras. In the compiler Bernoulli [6], P. Stoghill uses a generalization of several sparse storage formats as CCS,CRS, J.D ... However, both works are mainly focused on the automatic conversion of sequential dense programs into semantically equivalent sequential sparse codes. Whereas, numerical programs have very important computational times. Thus, it is interesting to define a framework for automatically extracting the parallelism of such programs. In this paper, we propose the definition of a new method for the automatic parallelisation of sequential programs working on dense matrices for generating a parallel counterpart working on sparse matrices.

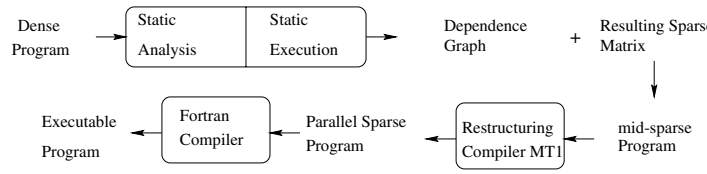
A sparse matrix contains many zero elements. This leads to define dedicated sparse storage formats to discard zero elements. However, it is not so straightforward to parallelize a program working on sparse storage formats. Indeed, programs using sparse storage format involves indirect addressing which inhibit symbolic analysis [4]. Thus, in order to parallelize programs with a dense data structure but operating on sparse matrices, we have to analyse dependencies by using the dense data structure. Finally, from the dependence graph computed, we deduce the parallel program. The main idea of our approach is to symbolically

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 340–348, 2000.
© Springer-Verlag Berlin Heidelberg 2000

compute dependencies from both the text program and the matrix in input by using the sparsity of matrices. Indeed, the sparsity of matrices leads to get more parallelism than dense matrices. This computation is split up into two steps:

1. computation of new entries, that is to say, positions in the matrix the content of which will become different of zero in the course of the numerical execution.
2. computation of iteration dependencies for generating the dependence graph. Here, we will use the previous step to refine usual dependence tests essentially based on the Bernstein's conditions.

The compilation scheme can be sketched as follows:



This article is focused on the top part of this compilation line. It is organized as follows: in Section 2, we define the filling function which from a program and a matrix in input computes the new entries. In Section 3, we generate the iteration sparse dependence graph by using results of the filling function defined in the previous section.

By lack of space, no proof of theorems is given in this paper. However, all these proofs can be found in the preliminary version of this paper [2].

1 Working Context

For the sake of simplicity, we reduce the analysis to one array for storing the matrix in input. Thus, we suppose to have in input of our compilation line a sequential program the form of which is inductively generated from assignments of form both $v = exp$ and $A[exp_1, \ldots, exp_c] = exp$ where v is a scalar variable, A is an array variable, exp_i are integer expressions ($1 \leq i \leq c$), exp is an expression. Control statements are the sequence operator, and both conditional and DO-loop constructions.
Given a program P, the definition of the filling function as well as dependencies will be dependent on assignments of the form $A[exp_1, \ldots, exp_c] = exp$ contained in a DO-loop nest within the program. Thus, afterwards we will consider the following generic form of these assignments:
$A[f(I_1, \ldots, I_d)] = \mathcal{G}(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)])$ so that A is the array of values in $\mathcal{C}$ (the domain of the concrete semantics) used to store the dense matrix, $f : \mathbb{Z}^d \to \mathbb{Z}^c$ (resp. each $g_p : \mathbb{Z}^d \to \mathbb{Z}^c$ for every $1 \leq p \leq m$) is an affine application yielding the index of a memory cell by writing (resp. by reading) from an iteration $(i_1, \ldots, i_d) \in \mathbb{Z}^d$, and $\mathcal{G} : \mathcal{C}^m \to \mathcal{C}$ stands for an application without side-effect.

More precisely, the application $\mathcal{G}$ is the semantical meaning in the standard interpretation $\mathcal{C}$ of the numerical expression $\mathcal{G}(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)])$ inductively generated from the following set of numerical operators:

operators	Definition
c	constant
v	variable
$\otimes$	binary operator so that 0 is absorbing
$\oplus$	binary operator so that 0 is neutral at left and/or at right
μ	unary operator or function for which 0 is a fixpoint (e.g. square root)
$\odot$, $\bar{\mu}$	both binary and unary operators or functions the behaviour of which depends on arguments (e.g. the randomize function)

We note $Expr$ the whole set of numerical expressions as described above. Finally, we note $Prog$ the whole set of well-formed programs which contains at least a DO-loop nest with a statement of the form:
$A[f(I_1, \ldots, I_d)] = \mathcal{G}(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)])$.

2 Symbolic Analysis

Herein, we describe how to generate at compile-time a symbolic program, called *filling function*, from a given numerical program. The goal of this symbolic program is to compute the fill-in introduced during the numerical computation. The fill-in deals with situations in which zero elements become nonzero. Well-known applications such as sparse Cholesky factorization use such symbolic analysis [5]. The difference from a program to another relies upon the definition of the "filling function" which is unique.

As we propose to generate the filling function at compile-time, the fill-in have to be derived from both the program text and the dense matrix in input. Usually, to statically collect dynamic informations about programs, it is natural to use a non-standard semantics of the programming language. The interest of such semantics is to abstract away from irrelevant matters by giving conservative approximations of the concrete behaviours of programs.

To compute the fill-in, we will use the elementary abstract interpretation theory by reinterpreting numerical expressions in the abstract domain $\mathbb{B} = \{true, false\}$ provided with the usual propositional connectors (principally $\wedge$ and $\vee$). Roughly speaking, given an assignment $A[exp_1, \ldots, exp_c] = exp$, "$true$" will mean that the evaluation of exp in the concrete semantics yields a value different of 0. Thus, the expression $exp_1, \ldots, exp_c$ will denote an *entry*, that is to say, an index the content of which is different of 0 in the course of the numerical execution. Succinctly, the idea is to use this abstraction to define an endofunction (the filling function) directly from the index space (not anymore from the iteration space of the program under analysis). Thus, we abstract ourselves of numerical execution. Then, we will be able to statically generate the set of new entries.

Afterwards, we choose to give as example a simplified version of of the Cholesky factorization algorithm. It corresponds to the code obtained after dropping all statements in Cholesky factorization that do not cause fill.

```
SPARSE, REAL ::A(N,N)
b1  do (j=1,N)
b2     do (k=1,j-1)
b3        do (i=j, N)
s            A(i,j) = A(i,j)-A(i,k)*A(j,k)
          enddo
       enddo
    enddo
```

The statement s belongs to a triple loop. Thus, the application $\mathcal{G} : \mathbb{R}^3 \rightarrow \mathbb{R}$ is defined by: $(x, y, z) \mapsto x - y * z$. Finally, the affine functions f, g_1, g_2, and g_3 from $\mathbb{N}^3$ to $\mathbb{N}^2$ are respectively defined by: $(i, j, k) \mapsto (i, j)$, $(i, j, k) \mapsto (i, j)$, $(i, j, k) \mapsto (i, k)$, and $(i, j, k) \mapsto (j, k)$.

2.1 Abstraction Domain

Notation 1. *Given a c-dimensional matrix, there exists a tuple* $(m_1, \ldots, m_c) \in (\mathbb{N}^+)^c$ *so that the underlying array* A *used to stock it is of size* $(m_1 \times \ldots \times m_c)$. *We note* $\mathbb{A}$, *so-called the* index space *of* A, *the set* $\{0, \ldots, m_1 - 1\} \times \ldots \times \{0, \ldots, m_c - 1\}$.

Definition 1. Let $\mathcal{C}$ be the domain where the standard interpretation of numerical expressions is defined (e.g. natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, real numbers $\mathbb{R}$, etc.). We defined the abstraction relation $\delta \subseteq \mathcal{C} \times \mathbb{B}$ by: $\delta = \{(0, true), (0, false)\} \cup \{(x, true) \mid x \neq 0\}$.

Remark. Understandably, $exp\ \delta\ true$ means that the expression exp provides a value which differs from zero. In this context, we can notice that zero is both linked with $true$ and $false$. This comes from the fact that some statements have their behaviour which is strongly dependent on the execution. For instance, facing an assignment of the form $A[f(I_1, \ldots, I_d)] = v$ where v is a scalar variable, we cannot statically deduce if the index denoted by the expression $f(I_1, \ldots, I_d)$ will be an entry or not. It depends on the value that v will have. Then, it is sensible to consider that the value of v is always different of 0.

As usual, expressions are evaluated from *environments*. Thus, given any domain $\mathcal{D}$, an environment $\rho_{\mathcal{D}}$ will associate the array A with an element of $[\mathbb{A} \rightarrow \mathcal{D}]$, [1] a variable v with an element of $\mathcal{D}$, and each iteration indice I with an element of its iteration space.

Definition 2. Given an environment $\rho_{\mathbb{B}}$ and a numerical expression exp of $Expr$, we note $[\![exp]\!]_{\rho_{\mathbb{B}}}$ the interpretation of exp in $\mathbb{B}$ inductively defined by the following rules:

- $[\![c]\!]_{\rho_{\mathbb{B}}} = (c \neq 0), [\![v]\!]_{\rho_{\mathbb{B}}} = true$, and $[\![A[g_p(I_1, \ldots, I_d)]]\!]_{\rho_{\mathbb{B}}} = \rho_{\mathbb{B}}(A)(g_p(\rho_{\mathbb{B}}(I_1), \ldots, \rho_{\mathbb{B}}(I_d)))$.
- $[\![exp_1 \otimes exp_2]\!]_{\rho_{\mathbb{B}}} = [\![exp_1]\!]_{\rho_{\mathbb{B}}} \wedge [\![exp_2]\!]_{\rho_{\mathbb{B}}}$.

[1] Given two sets N et M, the notation $[N \rightarrow M]$ denotes the whole set of applications from N to M.

- $[\![exp_1 \oplus exp_2]\!]_{\rho_{\mathbb{B}}} = [\![exp_1]\!]_{\rho_{\mathbb{B}}} \vee [\![exp_2]\!]_{\rho_{\mathbb{B}}}$.
- $[\![\mu(exp)]\!]_{\rho_{\mathbb{B}}} = [\![exp]\!]_{\rho_{\mathbb{B}}}$.
- $[\![\bar{\mu}(exp)]\!]_{\rho_{\mathbb{B}}} = true$.
- $[\![exp_1 \odot exp_2]\!]_{\rho_{\mathbb{B}}} = true$.

Remark. By Definition 2, both operations $\bar{\mu}$ and $\odot$ have to be interpreted as the constant function defined from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B}$ by: $(x, y) \mapsto true$.

Notation 2. *Given an environment $\rho_{\mathcal{C}}$ for the standard domain $\mathcal{C}$ and an environment $\rho_{\mathbb{B}}$, we say that $\rho_{\mathbb{B}}$ is* compatible with *$\rho_{\mathcal{C}}$ if and only if for every variable v we have: $\rho_{\mathcal{C}}(v)\delta\rho_{\mathbb{B}}(v)$ (δ being a relation).*

Proposition 3. *For every $\rho_{\mathcal{C}}$ and every $\rho_{\mathbb{B}}$ compatible with $\rho_{\mathcal{C}}$, we have: $[\![exp]\!]_{\rho_{\mathcal{C}}} \delta [\![exp]\!]_{\rho_{\mathbb{B}}}$.*

Proposition 3 establishes the correctness of the abstract interpretation.

2.2 Calculation of the Filling Function

In this section, by using the abstraction interpretation given in Section 2.1 we show how to statically generate the whole set of new entries. To reach this purpose, the idea is not to iterate anymore on iterations of DO-loop nests but on the entries themselves for generating new ones.

Notation 3. *Let $\rho_{\mathcal{D}}$ be an environment ($\mathcal{D}$ is any domain). Let v be any variable. An environment $\rho'_{\mathcal{D}}$ is* v-equivalent *to $\rho_{\mathcal{D}}$ if and only if $\rho'_{\mathcal{D}}$ is defined as $\rho_{\mathcal{D}}$ except for v.*

Definition 4. Given an environment $\rho_{\mathbb{B}}$ and a program P of *Prog*, we note $[\![P]\!]_{\rho_{\mathbb{B}}}$ the subset of $\mathbb{A}$ inductively defined by the following rules:

- $[\![v = exp]\!]_{\rho_{\mathbb{B}}} = \emptyset$.
- $[\![A[f(I_1, \ldots, I_d)] = exp]\!]_{\rho_{\mathbb{B}}}$ is the set of entries e so that for each one, there exists a tuple $(i_1, \ldots, i_d)$ of the iteration space so that both following conditions hold:
  - $e = f(i_1, \ldots, i_d)$.
  - for the environment $\rho'_{\mathbb{B}}$ I_j-equivalent to $\rho_{\mathbb{B}}$ with $\rho'_{\mathbb{B}}(I_j) = i_j$ for every $j = 1, \ldots, d$, we have: $[\![exp]\!]_{\rho'_{\mathbb{B}}} = true$.
- $[\![S_1; S_2]\!]_{\rho_{\mathbb{B}}} = [\![S_1]\!]_{\rho_{\mathbb{B}}} \cup [\![S_2]\!]_{\rho'_{\mathbb{B}}}$.
- $[\![if\ exp\ then\ S1\ else\ S2]\!]_{\rho_{\mathbb{B}}} = [\![S_1]\!]_{\rho_{\mathbb{B}}} \cup [\![S_2]\!]_{\rho_{\mathbb{B}}}$.
- $[\![do\ (I = P, Q)\ S]\!]_{\rho_{\mathbb{B}}} = [\![S]\!]_{\rho_{\mathbb{B}}}$

Notation 4. *Given an environment $\rho_{\mathcal{C}}$ (resp. $\rho_{\mathbb{B}}$), we note $E_{\rho_{\mathcal{C}}}$ (resp. $E_{\rho_{\mathbb{B}}}$) the subset of $\mathbb{A}$ defined by: $E_{\rho_{\mathcal{C}}} = \{e \mid \rho_{\mathcal{C}}(A)(e) \neq \emptyset$ (resp.$\{e \mid \rho_{\mathbb{B}}(A)(e) = true\}$) .*

Example 1. From the statement s_1 of the Cholesky algorithm and a given environment $\rho_{\mathbb{B}}$, we obtain for $[\![A(i,j) = A(i,j) - A(i,k) * A(j,k)]\!]_{\rho_{\mathbb{B}}}$ the following set of entries:

$$\begin{array}{l}\{(i,j) \mid \exists (j,k,i) \in \mathbb{Z}^3, \exists (x_0,y_0) \in E_{\rho_{\mathcal{C}}}, \exists (x_1,y_1) \in E_{\rho_{\mathcal{C}}}, \exists (x_2,y_2) \in E_{\rho_{\mathcal{C}}}, \\ \quad ((i,j) \notin E_{\rho_{\mathcal{C}}} \wedge (1 \leq j \leq N) \wedge (1 \leq k \leq j-1) \wedge (j \leq i \leq N) \\ \quad \wedge (((x_0 = i) \wedge (y_0 = j)) \vee (x_1 = i \wedge y_1 = k \wedge x_2 = j \wedge y_2 = k))\}\end{array}$$

As the constraints $1 \leq x_1, y_1, x_2, y_2 \leq N$ are always verified (the entry coordinates are limited to the matrix bounds) the characteristic function of the set $[\![A(i,j) = A(i,j) - A(i,k) * A(j,k)]\!]_{\rho_{\mathbb{B}}}$ can be simplified as follows:

$$\begin{array}{l}\{(x_1,x_2) \mid \exists (x_1,y_1) \in E_{\rho_{\mathcal{C}}}, \exists (x_2,y_2) \in E_{\rho_{\mathcal{C}}} \\ \quad (x_1,x_2) \notin E_{\rho_{\mathcal{C}}} \wedge y_1 = y_2 \wedge y_1 < x_2 \leq x_1\}\end{array}$$

Such simplifications are automatically performed by using symbolic computation tools as Omega [7]. This definition is not efficient as the handwritten code because the handwritten code exploits the transitivity to optimize fill computation.

Definition 5. With the previous notations, from a program P and an environment $\rho_{\mathbb{B}}$ which denotes the initial environment, we note $fill : 2^{\mathbb{A}} \rightarrow 2^{\mathbb{A}}$ where $2^{\mathbb{A}}$ is the set of all subsets of $\mathbb{A}$ (i.e. $2^{\mathbb{A}} = \{X \mid X \subseteq \mathbb{A}\}$), the application defined by: $\emptyset \mapsto E_{\rho_{\mathbb{B}}}$ and $E \mapsto E \cup [\![P]\!]_{\rho'_{\mathbb{B}}}$ where $\rho'_{\mathbb{B}}$ is any environment so that $E_{\rho'_{\mathbb{B}}} = E$.

To show that this application fully describes an algorithm, we use a classical result of set theory: the Tarski's theorem. Indeed, $(2^{\mathbb{A}}, \subseteq)$ is a complete partial order ($\emptyset$ is the least element and for any directed subset E, the upper bound $Sup\ E = \bigcup_{e \in E} e$). Moreover, $fill$ is obviously monotone since $\mathbb{A}$ is finite. Then, $fill$ is continuous (indeed, we have: $\forall e \in E, e \subseteq Sup\ E$). By the Tarski's theorem, $fill$ has a least fixpoint, usually noted fix_{fill}. Consequently, our algorithm is inductively defined by: $E^0 = E_{\rho_{\mathbb{B}}}, E^{t+1} = fill(E^t)$.

By Definition 5, this algorithm stops whatever the program and the matrix in input (the worst case is bounded by the cardinality of the iteration space). In [1] we show by experiments that the cost of the filling program is significantly less than the theoritical bound. We still have to show that our algorithm generates all entries as performed by the numerical execution.

Theorem 6. *Given a program P of Prog and an environment $\rho_{\mathcal{C}}$, we have: $E_{[\![P]\!]_{\rho_{\mathcal{C}}}} \subseteq fix_{fill}$ where $[\![P]\!]_{\rho_{\mathcal{C}}}$ stands for the meaning of P in the environment $\rho_{\mathcal{C}}$*

3 Sparse Dependence Analysis

Dependence analysis consists of determining the tasks of a program not being able to be performed independently. Thus, two tasks can be performed in parallel if for any order in which they are performed, the result is the same.

In general, the problem of computing all dependencies at compile-time is undecidable. However, we have sufficient conditions introduced by Bernstein which ensure us such a result. These conditions consist of verifying that all statements of the program under analysis do not access in the same time an identical cell memory. Due to the lack of space, we are only interested by the most important of the flow-dependence (a complete study of Berstein'conditions is given in [2], [1]).

Herein, tasks represent iterations of DO-loop nests. By following Section 1, each Do-loop nest of the program under analysis has the following generic form:

$$\begin{array}{l}\underline{\textbf{do}}\ (\boldsymbol{I} = (P_1, \ldots, P_d), (Q_1, \ldots, Q_d)\\ \quad S : A[f(I_1, \ldots, I_d)] = \mathcal{G}(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)])\\ \underline{\textbf{enddo}}\end{array}$$

For such a DO-loop nest, the flow-dependence condition is expressed: let us note $S(i_1, \ldots, i_d)$ to mean that the assignment S is performed at the iteration $(i_1, \ldots, i_d)$. Let $R(S)$ (resp. $W(S)$) be the set of cell memories read (resp. written) by the statement S. Then, for any $(i_1, \ldots, i_d) \ll (j_1, \ldots, j_d)$ where $\ll$ denotes the lexicographical order on the iteration space and means the *execution order*, $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$ are *flow-dependent* if and only if $W(S(i_1, \ldots, i_d)) \cap R(S(j_1, \ldots, j_d)) \neq \emptyset$.

We propose to refine the Bernstein's conditions by using the properties of 0 to be absorbing and neutral. To reach this purpose, we will use the abstraction domain as well as the following relation:

Notation 5. *Given an expression exp and a subexpression exp′ of exp, we note exp[exp′/x] the expression obtained from exp by substituting all occurrences of exp′ by a fresh variable x.*

Definition 7. Let $(i_1, \ldots, i_d)$ be an iteration of the iteration space. We note $\mathcal{S}_{(i_1,\ldots,i_d)} \subseteq Expr \times Expr$ the binary relation defined by:

$$exp\ \mathcal{S}_{(i_1,\ldots,i_d)}\ exp' \text{ iff } \begin{cases} \text{either } exp' \text{ is not a subterm of } exp, \\ \text{or for every } \rho_{\mathcal{C}} \text{ so that } \begin{cases} \rho_{\mathcal{C}}(I_j) = i_j,\ 1 \leq j \leq d \\ E_{\rho_{\mathcal{C}}} = fix_{fill} \end{cases} \\ \text{we have: } [\![exp]\!]_{\rho_{\mathcal{C}}} = 0 \qquad \text{if } exp = exp' \\ \qquad\qquad [\![exp[exp'/x]]\!]_{\rho'_{\mathcal{C}}} = [\![exp]\!]_{\rho_{\mathcal{C}}} \\ \qquad\qquad \text{for every } \rho'_{\mathcal{C}}\ x\text{-equivalent to } \rho_{\mathcal{C}} \text{ otherwise} \end{cases}$$

Let us suppose that exp has the form: $\mathcal{G}(A[g_1(I_1, \ldots, I_d)], \ldots, A[g_m(I_1, \ldots, I_d)])$, and exp' is of the form $A[g_p(I_1, \ldots, I_d)]$ where $1 \leq p \leq m$. Then, given an iteration $(i_1, \ldots, i_d)$, $exp\mathcal{S}_{(i_1,\ldots,i_d)}exp'$ means that the evaluation of exp does not depend on $A[g_p(i_1, \ldots, i_d)]$ whatever its content. For example, this condition holds when we are facing an expression $A[[g_p(i_1, \ldots, i_d)] \otimes A[g_{p'}(i_1, \ldots, i_d)]$ with $p' \neq p$ so that $g_{p'}(i_1, \ldots, i_d)$ is not an entry (i.e. $g_{p'}(i_1, \ldots, i_d) \notin fix_{fill}$).
We refine the flow-dependence condition in order to compute the sparse one.

Definition 8. With the previous notations, given two iterations $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$ so that $(i_1, \ldots, i_d) \ll (j_1, \ldots, j_d)$, we have $(i_1, \ldots, i_d)$ is *flow-dependent to* $(j_1, \ldots, j_d)$, usually noted $(i_1, \ldots, i_d)\delta_s^f(j_1, \ldots, j_d)$, iff:

$$\begin{array}{c} f(i_1,\ldots,i_d) \in fix_{fill} \\ \wedge\ (\exists 1 \leq p \leq m,\ f(i_1,\ldots,i_d) = g_p(j_1,\ldots,j_d)) \\ \wedge\ (\mathcal{G}(A[g_1(I_1,\ldots,I_d)],\ldots,A[g_m(I_1,\ldots,I_d)]), A[g_p(I_1,\ldots,I_d)]) /\!\in \mathcal{S}_{(j_1,\ldots,j_d)} \end{array}$$

Generating sparse dependencies at compile-time requires that the complement of the relation $\mathcal{S}_{(i_1,\ldots,i_d)}$ with respect to $Expr \times Expr$ be algorithmically definable. As for the filling function, we need to use the abstract interpretation defined in Section 2.1.

Definition 9. For any $(i_1,\ldots,i_d)$, let us note $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)} : Expr \times Expr \to \mathbb{B}$ the application inductively defined by:

- $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp', exp') = [\![exp']\!]_{\rho_{\mathbb{I}}}$ where $\rho_{\mathbb{I}}$ denotes any environment so that $E_{\rho_{\mathbb{I}}} = fix_{fill}$ and for every $j \in \{1,\ldots,d\}$ $\rho_{\mathbb{I}}(I_j) = i_j$.
- $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp, exp') = false$ if exp' is not a subterm of exp.
- $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_1 \otimes exp_2, exp') = [\![exp_1 \otimes exp_2]\!]_{\rho_{\mathbb{I}}} \wedge \begin{pmatrix} \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_1, exp') \\ \vee \\ \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_2, exp') \end{pmatrix}$
  where $\rho_{\mathbb{I}}$ denotes any environment so that $E_{\rho_{\mathbb{I}}} = fix_{fill}$ and for every $j \in \{1,\ldots,d\}$ $\rho_{\mathbb{I}}(I_j) = i_j$.
- $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_1 \mathbin{@} exp_2, exp') = \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_1, exp') \vee \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_2, exp')$
  where $@ \in \{\oplus, \odot\}$
- $\overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(@(exp_1), exp') = \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp_1, exp')$ where $@ \in \{\mu, \bar{\mu}\}$.

With such an approach, we only give a rough estimate of the complement of $\mathcal{S}_{(i_1,\ldots,i_d)}$ as it is shown by the following result:

Theorem 10. $(exp, exp') /\!\in \mathcal{S}_{(i_1,\ldots,i_d)} \Longrightarrow \overline{\mathcal{S}}_{(i_1,\ldots,i_d)}(exp, exp')$

From there, we can redefine iteration dependencies in such way that they can be automatically generated.

$$\overline{\mathcal{S}}_{(j_1,\ldots,j_d)}(\mathcal{G}(A[g_1(I_1,\ldots,I_d)],\ldots,A[g_m(I_1,\ldots,I_d)]), A[g_p(I_1,\ldots,I_d)])$$

Example 2. Due to the lack of space, we only present the analysis for the flow-dependencies. Two flow-dependencies are going to be computed from we will only give computations for $(\delta^{\mathbf{f}}_{\mathbf{s}})_{\mathbf{1}}$: defined from both $A(i,j)$ and $A(i',k')$.

$j(\delta^f_s)_1 j' \equiv$	$\exists (x_1,y_1) \in fix_{fill}, \exists (x_2,y_2) \in fix_{fill}, \exists (k,i,k',i') \in \mathbb{Z}^4,$
Domain by writing	$1 \leq j \leq N \wedge 1 \leq k \leq j-1 \wedge j \leq i \leq N \wedge$
Domaine by reading	$1 \leq j' \leq N \wedge 1 \leq k' \leq j'-1 \wedge j' \leq i' \leq N \wedge$
Identical references	$x_1 = i = i' \wedge y_1 = j = k' \wedge$
Sequential order	$j < j' \wedge$
$\overline{S}_{(i',j',k')}$	$i' = x_1 \wedge k' = y_1 \wedge j' = x_2 \wedge k' = y_2$

As previously, we can simplify the characteristic function as follows:

$$\begin{array}{l} j(\delta^f_s)_1 j' \equiv \exists (x_1,y_1) \in fix_{fill}, \exists (x_2,y_2) \in fix_{fill} \\ \qquad j = y_2 \wedge j' = x_2 \wedge y_2 = y_1 \wedge y_2 < x_2 \leq x_1 \end{array}$$

References

1. R. Adle : *Outils de parallélisation automatique des programmes denses pour les structures creuses*', PhD's thesis, University of Évry, 1999. In French. (ftp://ftp.lami.univ-evry.fr/pub/publications/reports/index.html)
2. R. Adle and M. Aiguier and F. Delaplace : *Automatic parallelization of sparse matrix computations: a static analysis*, Preliminary version appeared as Report LaMI-42-1999, University of Evry, december 1999. (ftp://ftp.lami.univ-evry.fr/pub/publications/reports/1999/index.html)
3. A.-J.-C. Bik and H.-A.-G. Wijshoff : *Automatic Data Structure Selection and Transformation for Sparse Matrix Computation*, IEEE Transactions on Parallel and Distributed Systems, vol. 7, pp. 1-19, 1996.
4. I.-S. Duff and A.-M. Erisman and J.-K. Reid : *Direct methods for sparse matrices*, Oxford Science Publications, 1986.
5. M. Heath and E. Ng and B. Peyton : *Parallel algorithm for sparse linear systems*, Siam Review, 33(3), pp. 420–460, 1991.
6. V. Kotlyar and K. Pingali and P. Stodghill : *Compiling Parallel Code for Sparse Matrix Applications*, SuperComputing, ACM/IEEE, 1997.
7. W. Pugh and D. Wonnacott : *An Exact Method for Analysis of Value-based Data Dependences*, Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, 1993.

Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition

Rashindra Manniesing[1], Ireneusz Karkowski[2], and Henk Corporaal[3]

[1] CWI, Centrum voor Wiskunde en Informatica,
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands,
r.manniesing@cwi.nl
[2] TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG Den Haag, The Netherlands,
karkowski@fel.tno.nl
[3] Delft University of Technology, Information Technology and Systems,
Mekelweg 4, 2628 CD Delft, The Netherlands,
h.corporaal@its.tudelft.nl

Abstract. This paper investigates the potential for automatic mapping of typical embedded applications to architectures with multimedia instruction set extensions. For this purpose a (pattern matching based) code transformation engine is used, which involves a three-step process of matching, condition checking and replacing of the source code. Experiments with DSP and the MPEG2 encoder benchmarks, show that about 85% of the loops which are suitable for *Single Instruction Multiple Data* (SIMD) parallelization can be automatically recognized and mapped.

1 Introduction

Many modern microprocessors feature extensions of their instruction sets, aimed at increasing the performance of multimedia applications. Examples include the Intel MMX, HP MAX2 and the Sun Visual Instruction Set (VIS) [1]. The extra instructions are optimized for operating on the data types that are typically used in multimedia algorithms (8, 16 and 32 bits). The large word size (64 bits) of the modern architectures allows SIMD parallelism exploitation. For a programmer however, task of fully exploiting these SIMD instructions is rather tedious. This because humans tend to think in a sequential way instead of a parallel, and therefore, ideally, a smart compiler should be used for automatic conversion of sequential programs into parallel ones.

One possible approach involves application of a programmable transformation engine, like for example the CTT (Code Transformation Tool) developed at the department CARDIT at Delft University of Technology [2]. The tool was especially designed for the source-to-source translation of ANSI C programs, and can be programmed by means of a convenient and efficient *transformation language*.

The purpose of this article is to show the capabilities and deficiencies of CTT, in the context of optimizations for the multimedia instruction sets. This has

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 349–356, 2000.
© Springer-Verlag Berlin Heidelberg 2000

been done by analyzing and classifying every FOR loop of a set of benchmarks manually, and comparing these results with the results obtained when CTT was employed.

The remainder of this paper is organized as follows. The CTT code transformation tool and the used SIMD transformations are described in detail in Sect. 2. After that, Sect. 3 describes the experimental framework, Sect. 4 presents the results and discussion. Finally, Sect. 5 draws the conclusions.

2 Code Transformation Using CTT

The CTT – a programmable code transformation tool, has been used for SIMD parallelization. The transformation process involves three distinct stages:

Pattern matching stage: In this stage the engine searches for code that has a strictly specified structure (that matches a specified pattern). Each fragment that matches this pattern is a candidate for the transformation.

Conditions checking stage: Transformations can pose other (non-structural) restrictions on a matched code fragment. These restrictions include, but are not limited to, conditions on data dependencies and properties of loop index variables.

Result stage: Code fragments that matched the specified structure and additional conditions are replaced by new code, which has the same semantics as the original code.

The structure of the *transformation language* used by CTT closely resembles these steps, and contains three subsections called PATTERN, CONDITIONS and RESULT. As can be deduced, there is a one-to-one mapping between blocks in the transformation definition and the translation stages. While a large fraction of the embedded systems are still programmed in assembly language, the ANSI C has become a widely accepted language of choice for this domain. Therefore, the transformation language has been derived from the ANSI C. As a result, all C language constructs can be used to describe a transformation. Using only them would however be too limiting. The patterns specified in the code selection stage would be too specific, and it would be impossible to use one pattern block to match a wide variety of input codes. Therefore the transformation language is extended with a number of *meta-elements*, which are used to specify generic patterns. Examples of meta-elements, among others, are the keyword STMT representing any statement, the keyword STMTLIST representing a list of statements (which may be empty), the keyword EXPR representing any expressions, etc. We refer to [2] for a complete overview, and proceed with a detailed example of a pattern specification.

Example of a SIMD Transformation Specification

The example given describes the *vectordot product loop* [1]. The *vectordot product loop* forms the innerloop of many signal-processing algorithms, and this particular example is used, because we base our experiments on this pattern and a number of its derivatives.

```
PATTERN{     VAR i,a,B[DONT_CARE],C[DONT_CARE];
             for(i=0; i<=EXPR(1); i++) {
                 STMTLIST(1);
                 MARK(1);
                 a+= B[i]*C[i];
                 STMTLIST(2);}}

RESULT{      VAR i;
             VAR a,B[DONT_CARE],C[DONT_CARE];            /* arrays of signed int.(16 bits)*/
             VAR bfl,cfl,bfh,cfh;                        /* Intermediate var.(2x16 bits)*/
             VAR bf,cf,ub,tuh,tlh,tul,tll,tdh,tdl,td,aa; /* Intermediate var.(2x32 bits)*/
             DEFINE_TYPE_FROM_STRING("ub", "int");
             DEFINE_TYPE_FROM_STRING("bfl", vis_f32_s);
             ...
             DEFINE_TYPE_FROM_STRING("aa", vis_d32_s);
             for(i=0;i<=EXPR(1);i++){STMTLIST(1);}
             ub=EXPR(1)/4;
             for(i=0;i<ub;i++){
                 bfh = *(vis_f32 *)(B+i*4);          /*load 4x16 bits signed*/
                 bfl = *(vis_f32 *)(B+i*4+2);        /*integers, into two steps*/
                 cfh = *(vis_f32 *)(C+i*4);          /*Results into variables of*/
                 cfl = *(vis_f32 *)(C+i*4+2);        /*2x16 bit (bfh,bfl,cfh,cfl)*/
                 tuh = vis_fmuld8sux16(bfh,cfh);     /*multiplies 8x16 and shift8*/
                 tlh = vis_fmuld8ulx16(bfh,cfh);     /*multiplies 8x16, no shift*/
                 tul = vis_fmuld8sux16(bfl,cfl);     /*for higher (bfh,cfh) and for */
                 tll = vis_fmuld8ulx16(bfl,cfl);     /*lower (bfl,cfl). Results 2x32 bits*/
                 tdh = vis_fpadd32(tuh,tlh);         /*2x32 bits additions*/
                 tdl = vis_fpadd32(tul,tll);
                 td = vis_fpadd32(tdh,tdl);
                 aa = vis_fpadd32(aa,td);            /*Final result (2x32 bits)*/}
             a = getL(aa)+getH(aa);
             for(i=4*ub;i<=EXPR(1);i++){a=a+B[i]*C[i];}
             for(i=0;i<=EXPR(1);i++){STMTLIST(2); }}

CONDITIONS{  var_is_type(a,"long int"),var_is_type(B,"int []"),var_is_type(C,"int []");
             expr_is_constant(1);
             not(dep("true DISTANCE>=(1) between stmtlist 2 and stmtlist 1"));
             not(dep("true DISTANCE>=(1) between mark 1 and stmtlist 1"));
             not(dep("true DISTANCE>=(1) between stmtlist 2 and mark 1"));}
```

Fig. 1. Vectordot product pattern with reduction

Figure 1 shows the specification in which some details have been left out (for example inclusion of the header files in the beginning). The specification starts with the search pattern description. In there, the FOR loop (used for matching) assumes well defined boundaries. These can be obtained by applying the preprocessing step which normalizes all FOR loops of the source code. Within the loop body, we can see two statement lists, a statement and a `MARK` meta-element. The statement will match with the multiplication of two 16 bits signed integers, while the accumulator variable `a`, must be 64 bits long. This is an example of a statement with *reduction*: reduction refers to an accumulator variable in the expression within the loop body. The `MARK` meta-element is used to refer later to the statement itself (from within the condition block).

The result block starts with the creation and definitions of types of intermediate variables. Some of them have been left out, to prevent the figure becoming

too large. After that the first, third and fourth FOR loops handle the statement lists 1 and 2, and the remaining iterations of the second loop (`ub` modulo 4). The most interesting part is of course the second loop. It implements the SIMD parallelization of the statement from the pattern block.

The condition block checks the upper-bound of the loop-index `EXPR(1)` and dependencies between different parts within the loop body. The upper-bound must be a constant, and no dependencies from `STMTLIST(2)` to `STMTLIST(1)` (and the other two) are allowed.

Note that this transformation is actually a combination of two simpler transformations. The first one is the well known loop fission [3] (which allows us to handle loops which have more than one statement). The second one converts a simple statement loop into a SIMD loop.

Some remarks are in order. For simplicity of presentation we ignored the problem of unaligned arrays (the extension is straightforward [4]). Secondly, the above transformation will work only for arrays with elements of type "int". Very similar transformations may be written for other basic types. We ignored the problem of statement lists being empty. It is not serious – the post-processing passes may be used to remove loops with empty bodies (alternatively we could write separate transformations for these cases). Finally, a parallel loop may contain several statements suitable for mapping onto SIMD instructions. To exploit this potential the above transformation (and the others) should be applied repeatedly until no more candidates are found.

3 Experimental Framework

From this pattern similar ones have been derived to form a class of patterns to search for by CTT in the experiments. We used two types of transformations for SIMD parallelization, one with reduction in the loop body and one without. Furthermore, within each type, the pattern block differs in the operator which results in a total of 8 different patterns (we consider $+, -, *, /$ operators only).

To make a successful parallelization possible, a number of pre-processing steps need to be applied on the source code [2]. These steps involve the following: the first step flattens expression trees inside loops. It will break up long expressions by introducing temporary variables. The next step normalizes all loops, resulting in uniform index descriptions. Finally, the third step expands scalars into arrays inside loops. The last step is necessary to make loop fission [3] (being part of each transformation; recall section 2) legal. Loop fission allows us to handle loops containing more than one statement.

From those steps, only the second one has been applied, because the front-end SUIF trajectory [5], which we use, did not support the others. Unfortunately, in order to determine the potential for SIMD parallelization we do need these steps. Instead, we used patterns which have very general expressions. For example, the multiplication expression (the statement from transformation example in Fig. 1) became `a=a+EXPR(1)*EXPR(2)`. This relaxation is possible because our purpose was to estimate the number of loops that can be automatically parallelized and

Table 1. Benchmarks characterization ('r' – with reduction)

Bench-marks	Description	FOR loops	Outer-loops	SIMD	Non-SIMD *Fn*	*In*	*Cm*	*Dp*	CTT matches *add*	*sub*	*mul*	*div*	SIMD map
arfreq	Autoregr. freq. estim.	2	0	0		1		1	2	1	1r		0
g722	Adaptive diff. PCM	12	0	7		4	1		4	2	1,2r		6
instf	Frequency tracking	9	1	3	1	2	1	1	5		1,2r		3
interp3	Sample rate conversion	3	0	0	1			2	1				0
mulaw	Speech compression	1	0	0	1				1				0
music	Music synthesis	4	1	0	1	1		1	2			1	0
radpr	Doppler radar proc.	7	2	1	2		1	1	4	2	1,1r		1
rfast	Fast FTT convolution	9	0	5	3	1			3	1	1		4
rtpse	Spectrum analysis	10	2	3	3	1		1	3	1	2,1r		3
mpeg2	Video/MPEG2-enc	171	57	26	21	8	34	25	51,1r	15	1,12r	4	22
Total		228	63	45	33	18	37	32	76,1r	22	7,19r	5	39

to compare this number with the number of loops which are actually suitable for the SIMD parallelization. The results obtained this way will be summarized in one table in the next section.

4 Results and Discussion

In our experiments we used two sets of benchmark files – the DSP benchmarks [6] and the MPEG2 Encoder [7] benchmark. They consist of 9 and 15 files, and have a total number of 57 and 171 FOR loops, respectively. All loops have been individually classified. Table 1 summarizes their most important characteristics.

Concentrate on Table 1. After the file's name, the number of FOR loops it includes is denoted, followed by the column *outer-loop*. A loop is defined as an outer-loop if it contains another FOR loop in its body, but without any other statements. Clearly an outer-loop has no use for parallelization because its body contains nothing else but another FOR loop. In all benchmarks, the maximum depth of nested loops did not exceed two.

The column "SIMD" denotes the number of FOR loops which should be suitable for SIMD parallelization (according to manual inspection). The remaining loops were not suitable for SIMD. Their numbers are captured in the column "Non-SIMD", and are classified into the following categories:

- *Fn* (function) – the body has a function call or procedure call.
- *In* (init) – the loop initializes some variables by setting them to fixed values.
- *Cm* (compare) – IF statement or a SWITCH statement has been used.
- *Dp* (dependency) – there is inter-iteration dependency in the loop body.

Note that this is not an exclusive classification. For example, a FOR loop may simultaneously not be suitable for SIMD because of dependencies (depend +1) and possibly because of a case statement in the loop body. In the table, only

one classification will then be made. This would be 'Cm', because compare (as well as 'In' and 'Fn') allows some parallelization with the right patterns or preprocessing steps, unlike the classification 'Dp'. In other words, the classification 'Dp' has always the highest priority. Following above general rule leads to an easy check-up function within the table: summation of outer-loops, SIMD and non-SIMD should be equal to the total number of FOR loops.

The remaining columns denote the actual results obtained by running CTT ('r' in the table is 'reduction'), of which the last column might be the most interesting as it directly shows how well CTT performs with this particular transformation library of 8 patterns (the "SIMD map" results are manually verified).

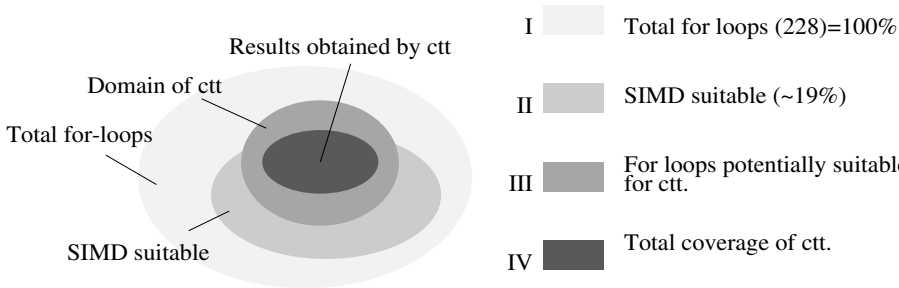


Fig. 2. Diagram of all FOR loops from the benchmarks

Discussion

At first glance, there seems to be some contradictions within the table. For example, the first benchmark *arfreq* has no loops which are suitable for parallelization, while according to "CTT matches" in the table, four pattern matches were found when running CTT. The reason is that the experiments only presents the results of the code selection stage: it shows how well CTT performs in pattern matching and describes, for a valid FOR loop, which patterns need to be applied for parallelization. Another problem can arise, when reading the table. For example, the *g722* file has 7 FOR loops which are suitable for parallelization, while CTT finds a total number of 9 matches. This is caused by the multiple matches within the same (multi-statement) loop body.

Consider a graphical overview of all the results, presented in Fig. 2, which illustrates the domain of all FOR loops. This domain consists of four different regions: Region I, the most light-gray circle, shows a total number of 228 FOR loops (100%). From these, approximately 19% are found (by manual inspection) suitable for SIMD mapping (region II). Region III includes all the loops which CTT should be able to find (also non-SIMD), and region IV denotes the actual results obtained by CTT.

The part of region I outside of region II presents all the loops not suitable for SIMD mapping. It includes the loops classified in the table as 'Dp' (depend, 14%) and 'outer-loop' (28%). The other loop categories ('Fn'-function, 'In'-initialization and 'Cm'-compare) have a certain number of FOR loops which could possibly belong to the domain of CTT. Therefore region III (domain of CTT) covers part of the FOR loops outside region II.

Region II represents the loops suitable for SIMD mapping and has a large potential for CTT to exploit. In [1] four widely used algorithms are described, which should benefit from VIS instructions: the *separable convolution, sum of absolute differences, trilinear interpolation* and *the vectordot product*. The 8 patterns, which we use, are all derived from the *vectordot product*. The other three algorithms are not covered at all. This explains the part of region II not covered by region III. As a consequence, the region III has two ways to expand: first, by writing the patterns and all its derivatives, specific for the other three algorithms, resulting in a larger coverage of region II. And second, by handling the loops classified as Fn/In/Cm in region I, which can result in larger coverage of region I.

Speedup As could be seen in both previous sections the coverage of CTT is reasonable (approximately 85% of SIMD suitable loops are recognized). However, if we take into account that the region II represents only 19% of the total number of loops, the question arises if the SIMD parallelization obtained this way is worth the effort. The answer to this question very much depends on the benchmark in question. The final speedup will namely depend on the fact if we are able to parallelize the most frequently executed parts of a given benchmark. This speedup may be calculated using the following formula:

$$s = \frac{L_{total}}{L_{total} - \sum_{i \in P} L_i + \sum_{i \in P} L_i / s_i}$$

where L_{total} is the total sequential execution time of the benchmark, P the set of parallelized loops, L_i sequential latency of parallelized loop i and s_i local speedup obtained in loop i.

As an example consider the *instf* benchmark. One of its two most important parts is the routine *lms*, which includes 3 very frequently executed loops. Two of these loops are perfectly suitable for SIMD parallelization and are without problems parallelized by CTT. Since they both constitute about 40% of the total execution time of the benchmark, the obtained speedup[1] amounts to approximately 1.5.

Improvements Further we conclude the following:

- The inspection of the benchmarks shows that inter-procedural transformations as a pre-processing steps are justified (33 occurrences).

[1] Overhead of SIMD approach depends on the processor SIMD support and can be substantial.

- In the benchmarks, initializations of variables within a loop (that is initialization at zero, or one-to-one copy of another variable or array) does occur often (18 occurrences) pleading for parallelizing them as well. Especially, because these transformations are simple to write.
- From Table 1, we learn also that most matches occur with the addition expression (59% of total matches). The pattern library should therefore at least contain addition transformations for various types of the variables and/or arrays.
- Expressions with reduction are in minority compared to expressions without reduction (19+1=20 and 76+22+7+5=110, respectively). A suggestion is to break the first type of expressions into several ones (another atomization pre-processing step), limiting this way the size of the transformation library.

5 Conclusions

In this paper we investigated the potential for automatic SIMD parallelization of embedded applications. For this purpose a programmable (pattern matching based) code transformation engine was used. In our experiments we were able to automatically recognize and map about 85% of the loops which were suitable for SIMD mapping. While this number is quite high, in general large coverage does not guarantee the overall speedup in the application. This speedup depends also on the execution time profile, which is independent from the number of SIMD suitable loops. While clearly there exists a limit on the number of loops which can be automatically parallelized [1], increasing the coverage of an automatic SIMD parallelizer is certainly advantageous. The extension of the inter-procedural transformations, has been identified as the most promising direction.

References

1. Marc Tremblay et al. Vis speeds new media processing. *IEEE micro*, August, 1996.
2. Maarten Boekhold, Ireneusz Karkowski, and Henk Corporaal. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. In *HPCN Europe'99*, Amsterdam, NL, April 1999.
3. Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.
4. Gerald Cheong and Monica S. Lam. An Optimizer for Multimedia Instruction Sets. In *Proceedings of the Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.
5. Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shin-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary W. Hall. Multiprocessors From a Software Perspective. *IEEE micro*, pages 52–61, June 1996.
6. P.M. Embree. *C Language Algorithms for Real-Time DSP.* Prentice-Hall, 1995.
7. MPEG Software Simulation Group, http://www.mpeg.org/index.html/MSSG/#source. *MPEG-2 Video Codec*, 1996.

Temporary Arrays for Distribution of Loops with Control Dependences

Alain Darte[1] and Georges-André Silber[2]

[1] LIP, ENS-Lyon, 46, allée d'Italie, 69007 Lyon, France.
[2] CRI, ENSMP, 33, rue Saint Honoré, 77305 Fontainebleau Cedex, France.

Abstract. We consider the problem of distribution of loops with control dependences, involving if and do control structures. More precisely, we study how to control the number of temporary arrays that have to be introduced to store conditionals. We show that the traditional superposition of the data dependence graph and of the control dependence graph is not adequate, and we introduce a new representation, the mixed dependence graph. This allows us to develop a distribution algorithm that is parameterized by the maximal allowed dimensions of temporary arrays.

1 Introduction

Code transformations have not only to take into account data dependences (dependences between memory accesses) but also control dependences in the case of a complex control flow, as for instance if an expression contained in an if statement is involved in a data dependence. By complex, we mean that the execution of some statements is not known at compile-time: such codes are called *nonstatic control codes*. As we will see in this paper, the *reduced dependence graph* (RDG) usually used by parallelization algorithms or more generally code transformations does not represent this kind of code very well. Several approaches have been proposed to handle this problem. The first one, known as "*if conversion*", converts these control dependences to data dependences [3]. This approach systematically introduces a temporary array, modifying deeply the code, even if the temporary is actually not needed. Another approach, presented by McKinley and Kennedy [6], uses the control dependence graph in combination with the data dependence graph to (try to) introduce temporaries only when it is really necessary. However, we will see that this approach restricts the set of valid codes and does not allow the user to drive the introduction of temporary arrays.

We present a new method to handle control dependences thanks to what we call the *mixed dependence graph* (MDG). We consider only two code transformations in this paper: loop fusion and loop distribution [7]. To illustrate our method, we show how to use this graph to develop an extension of Allen, Callahan, and Kennedy's algorithm [2]. In the case of a static control code, this graph is nothing but the RDG containing only data dependences. In the case of a nonstatic control code, some dependences are added that summarize exactly the constraints that have to be respected. Moreover, the introduction of temporary

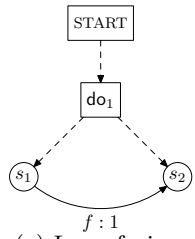
A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 357–367, 2000.
© Springer-Verlag Berlin Heidelberg 2000

arrays can be controlled during the search for a valid distribution partition: this was not the case in previous approaches.

2 Distribution of Control Structures: Related Works

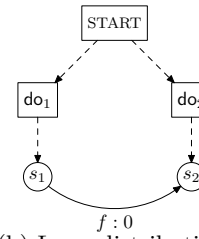
The distribution of loop converts a single loop to several ones, each containing a subset of the statements of the original loop body. This transformation has many interests in compilers [9,12], e.g., to reveal parallelism. The inverse transformation (loop fusion [1,8]) aggregates several compatible loops into one. Fig. 1 gives an example of loop distribution/fusion. Below each loop, we show the superposition of two graphs: the *reduced dependence graph* (RDG) representing data dependences (bold arrows) and the *control dependence graph* (CDG) [4] representing control dependences (dotted arrows). Square (resp. round) vertices represent control statements (resp. basic actions). Each data dependence has a type (f for flow, a for anti, and o for output) and a level (depth of the loop that carries the dependence, except that we use 0 for a loop-independent dependence). For example, the code on the left has a loop-carried flow dependence due to the references $a(i)$ and $a(i-1)$. After distribution, the dependence is not carried anymore and the two loops are parallel. We use doall for a parallel loop, with the semantics of the HPF independent and the OpenMP parallel directives.

```
do i = 2, n                    (* do1 *)
   a(i) = b(i) + c(i)          (* s1 *)
   d(i) = a(i - 1) + e(i)      (* s2 *)
enddo
```



(a) Loop fusion.

```
doall i = 2, n                 (* do1 *)
   a(i) = b(i) + c(i)          (* s1 *)
enddo
doall i = 2, n                 (* do2 *)
   d(i) = a(i - 1) + e(i)      (* s2 *)
enddo
```



(b) Loop distribution.

Fig. 1. Example of loop fusion/distribution with the associated dependence graphs.

When no data dependences involve control structures, the situation is clear. Considering the RDG alone (the bold arrows of our graphs), a loop distribution is legal if and only if (1) all statements involved in a data dependence circuit are in the same loop after distribution, (2) if, in the new code, there is a data dependence from a statement s_1 to a statement s_2 in a different loop, then

s_1 appears textually before s_2, and (3) if, in the new code, there is a loop-independent data dependence from a statement s_1 to a statement s_2 in the same loop, then s_1 is before s_2 in the loop body (initial textual order). A *valid partition* of a RDG is a set of disjoint groups of vertices that cover the RDG and such that each data dependence circuit is contained in a group. Each group represents a loop after distribution. A valid partition obviously enforces Condition (1). Condition (2) is respected if the loops are generated following a topological order defined by the arcs that cross groups. Condition (3) is naturally respected if the original textual order is respected during the code generation of each group/loop.

2.1 Complex Control Flow

In the presence of a nonstatic control code, loop distribution is more complicated. For instance, an if statement with an expression involved in a data dependence may prevent distribution: this is the case in Fig. 2. In this code, there is a data dependence from the expression in if_1 to the action s_1. If we apply a simple loop distribution, as for a static control flow code, we will place each elementary statement in a new control structure, a priori without duplication of memory, but with a duplication of the expression contained in the if (same thing for loop bounds). When there is an anti dependence as the one in Fig. 2, the evaluation of the expression contained in if_1 or $\mathsf{if'}_1$ is not the same whether it is done after or before s_1. In this case, if we distribute the loop, the resulting code is wrong.

```
(* Original code *)
do i = 2, n                      (* do1 *)
   if (a(i) == 0) then           (* if1 *)
      a(i) = b(i) + c(i)         (* s1 *)
      d(i) = a(i - 1) + e(i)     (* s2 *)
   endif
enddo
```

```
(* Semantically incorrect code *)
do i = 2, n                      (* do1 *)
   if (a(i) == 0) then           (* if1 *)
      a(i) = b(i) + c(i)         (* s1 *)
   endif
enddo
do i = 2, n                      (* do'1 *)
   if (a(i) == 0) then           (* if'1 *)
      d(i) = a(i - 1) + e(i)     (* s2 *)
   endif
enddo
```

Fig. 2. A code where simple loop distribution is forbidden.

To distribute the loop do_1, several approaches have been proposed, all sharing the same principle: introducing extra memory to store the evaluation of an expression for a control structure that is involved in a data dependence.

2.2 If Conversion

The first approach, known as "if conversion" [3], is to add an assignment statement inside the body of the loop to store, into a temporary memory of the

appropriate size, the evaluation of the expression defined in the control structure. Then, an evaluation of the temporary memory guards each statement in the control structure. Fig. 3 shows the if conversion of the code of Fig. 2.

```
do i = 2, n                                  (* do1 *)
    t(i) = (a(i) == 0)                       (* t1 *)
    if (t(i)) a(i) = b(i) + c(i)             (* s1 *)
    if (t(i)) d(i) = a(i − 1) + e(i)         (* s2 *)
enddo
```

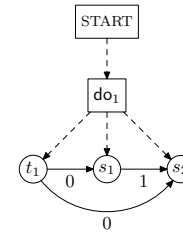


Fig. 3. Introduction of a temporary array for the code of Fig. 2.

This operation corresponds, in the dependence graph, to the conversion of a "control" dependence to a data dependence in the sense that no control structure is involved in a data dependence anymore. The term "control" here does not mean an actual control dependence but a data dependence involving a control structure. After if conversion, the loop can be distributed the usual way (see Fig. 4). Nevertheless, this approach has several drawbacks. First, it is a systematic approach that may introduce temporary memory even if it is not necessary, increasing the size of the memory used (a temporary memory can have the size of the full iteration space). Second, once if conversion is done, it is difficult to undo it: the code can be deeply modified. Finally, if an if statement encloses a loop, the if conversion "pushes" the if "down" as guarded statements, possibly modifying the number of iterations introducing iterations with no real computation inside.

```
doall i = 2, n                               (* do1 *)
    t(i) = (a(i) == 0)                       (* t1 *)
    if (t(i)) a(i) = b(i) + c(i)             (* s1 *)
enddo
doall i = 2, n                               (* do''1 *)
    if (t(i)) d(i) = a(i − 1) + e(i)         (* s2 *)
enddo
```

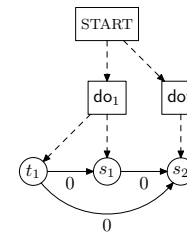


Fig. 4. Distribution of the loop thanks to the temporary array.

2.3 McKinley and Kennedy's Approach

A more recent approach, dedicated to loop distribution, was proposed by McKinley and Kennedy [6], as an attempt to introduce temporaries only when really

needed. The decision of introducing a temporary array is taken according to a given partition of the "full" dependence graph, i.e., the superposition of the data and control dependence graphs. This amounts to simulate what if conversion would do, but without pre-transforming the code. The code is transformed only *after* the partition is chosen. In this approach, control dependences are considered exactly as data dependences for the validity of the partition and there is no distinction between anti and flow dependences. Going back to Ex. 2, the partition that leads to the same code as Fig. 4 is the partition with if_1 and s_1 in one group, and s_2 in another group. A temporary array is introduced each time there is a control dependence that crosses two groups of the partition. The main point is that vertices representing an if structure are included in the groups: in this model, an if structure represents the evaluation of an expression and (possibly) its storage into a temporary array. Although this approach is an improvement over if conversion, it has several weaknesses. First, it does not allow the recomputation of an expression contained in an if or a do control structure as it is naturally done for static control codes, second, it cannot capture all valid codes.

Consider the code of Fig. 5. There is a data dependence from the expression in if_1 to the statement s_1. Imagine that we want to obtain the code on the right, with s_2 and s_3 in the same parallel loop, and s_1 in another sequential loop (and no temporary array). The three partitions of Fig. 6 are the only possible

```
do j = 2, n − 1                  (* do1 *)
   if (a(j + 1) == 0) then       (* if1 *)
      a(j) = b(j) + c            (* s1 *)
      d(j) = u(j) × e            (* s2 *)
   else
      d(j) = a(j + 1) + 1        (* s3 *)
   endif
enddo
```

```
doall j = 2, n − 1
   if (a(j + 1) == 0) then
      d(j) = u(j) × e
   else
      d(j) = a(j + 1) + 1
   endif
enddo
do j = 2, n − 1
   if (a(j + 1) == 0) then
      a(j) = b(j) + c
   endif
enddo
```

Fig. 5. An example where distribution is possible with no temporary array.

partitions where s_2 and s_3 are in the same group, and s_1 in another group. Partitions 1 and 3 are valid but a temporary array is needed because of the control dependence that crosses the groups. Partition 2 is not legal because of the circuit between the two groups. Therefore, with this approach, there is no partition corresponding to the previous code: a temporary array is always introduced.

The reason why McKinley and Kennedy's approach does not lead to all valid codes is twofold. First, when a control dependence crosses two groups, a tempo-

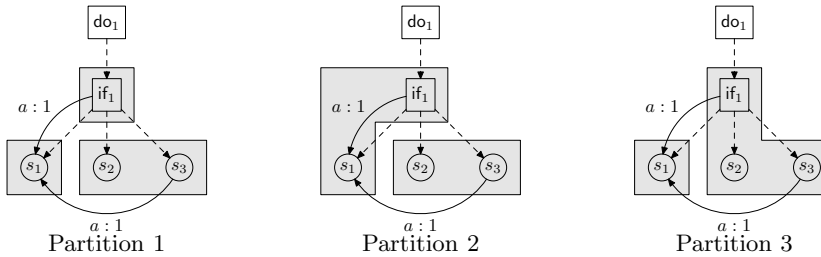


Fig. 6. Three partitions of the dependence graph of Fig. 5.

rary array is not always required: this is the case when the control dependence corresponds to an anti dependence and all guards are evaluated *before* the sink of the dependence. Second, defining the validity of a partition through the superposition of the data and control dependence graphs forbids, by nature, all codes where re-computation of expressions is performed. The formalism we present in the next section solves this problem. Furthermore, it allows us to search for a valid partition while controlling the introduction of the temporaries.

3 The Mixed Dependence Graph

We now present what we call the *mixed dependence graph* (MDG), which is an extension of the RDG in the case of nonstatic control codes. We first define the MDG when no temporary memory is introduced. Then, we present a way to modify the graph that takes the introduction of temporary arrays into account.

3.1 Definition

The problem with the superposition of the RDG and the CDG is that it mixes vertices of two different natures, control structures and elementary statements. For example, considering a control dependence from an if to an elementary statement as a dependence with the same nature as a data dependence is nothing but considering that the expression contained in the if is systematically stored and used. This is why codes with re-computation of expressions instead of storage cannot be expressed. The main idea of the MDG is to avoid this artificial superposition and to manipulate only vertices with a unique and clear semantics. The MDG is built from the CDG and the RDG as follows.

The MDG have as many vertices as elementary statements, but each vertex in the MDG represents not only a given elementary statement, but the whole *path* in the control dependence graph, from a root vertex S to the elementary statement. The vertex S allows us to concentrate to a portion of code at a given depth: if we consider a portion of code not contained in a control structure, S is the vertex START. Otherwise S is the control structure that contains the code.

(We restrict our study to codes with no goto statements to ensure that there is only one path from the vertex START to an elementary statement in the CDG.)

The arcs in the MDG are defined as follows: each data dependence in the RDG from a vertex u to a vertex v generates an arc in the MDG (keeping track of any information, depth of statements, level and type of dependence, etc.) from any path (i.e., a vertex in the MDG) containing u to any path containing v. In other words, the MDG represents what is needed to check if an elementary statement will execute and everything to compute it. In the MDG, we will consider two kinds of dependences: the regular data dependences that come from a dependence between two basic statements in the RDG and the *path dependences* that come from a data dependence involving an expression of a control structure.

The left part of Fig. 7 represents the MDG corresponding to the CDG/RDG of Fig. 5. It has one regular dependence (bold arrow), the anti dependence from s_3 to s_1, and 3 path dependences (dotted arrows) representing the data dependence from if_1 to s_1: one from the path containing s_1 to the path containing s_1 (a self-dependence), one from the path containing s_2 to the path containing s_1 (because if_1 is also contained in the path containing s_2), and one from the path containing s_3 to the path containing s_1 (same reason). Note that this dependence could have been a flow dependence but not an output dependence: we only consider a language where an expression defined in a control structure cannot modify a memory storage. The right part of Fig. 7 represents a legal partition of the MDG. Indeed, the conditions for a partition to be legal are the same than those given in Section 2. The MDG can now be used as a standard RDG: preserving all path dependences guarantees that no temporary arrays will be needed. Note also that, in the case of static control codes, the MDG *is* the RDG.

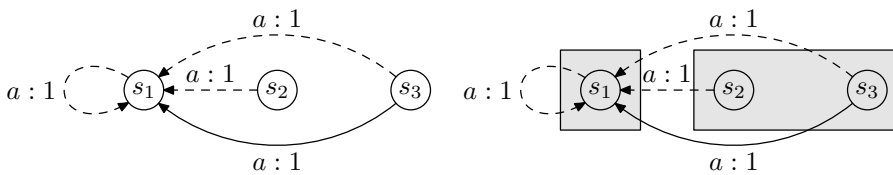


Fig. 7. On the left, the MDG for the first code of Fig. 5 (see the corresponding RDG and CDG on Fig. 6). On the right, a legal partition leading to the second code of Fig. 5.

Path dependences represent the fact that if a control expression (like in a do loop or an if) is the sink of a flow dependence, all its potential duplications by distribution must be executed **after** the modification of the expression by the source of the flow dependence. A path flow-dependence constrains valid codes by forbidding a distribution in the case of a circuit and makes the loop sequential if the dependence is carried. A path anti-dependence also constrains the valid codes: this time, the expression must be evaluated **before** its modification by

the sink of the anti dependence. When a distribution duplicates the expression, all loops that contain it must be computed **before** the loop containing the sink of the dependence, or, if the source and the sink are in the same loop, the original organization of these two statements, in terms of control, must be preserved.

3.2 Introducing Temporary Arrays

The previous MDG formulation does not integrate the possibility of introducing temporary arrays but it characterizes all valid codes that require no extra memory. Let us see now how to incorporate temporary arrays into the model. The idea is that any path anti-dependence can be suppressed by a temporary array of sufficient size and dimension, by introducing an assignment statement to store the expression at the right place and by reusing this temporary array in the loops (if any) that are executed **after** the loop that modifies the expression.

Let us go back to the example of Fig. 5. The path anti-dependence generated by the anti dependence from $a(j+1)$ to s_1 can be suppressed by a temporary array of dimension 1 (see Fig. 8 for the effect on the graphs). Note that in a

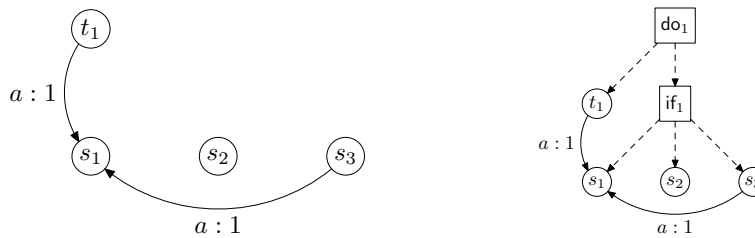


Fig. 8. MDG (and CDG/RDG) after the introduction of a temporary array.

standard if conversion, there would be a flow dependence from t_1 to if_1, and from t_1 to all uses of the temporary. Those dependences are not in the MDG because we do not know yet if these statements will use the temporary or simply recompute the expression. This decision will be taken during code generation depending on the partition. Indeed, consider a statement whose execution depends on the expression stored in the temporary. If this statement is generated before the temporary assignment, it **cannot** use the temporary array but should recompute the expression. If it is generated between the assignment expression and the sink of the dependence, it **can** use the temporary. And finally, if it is generated after the sink of the dependence, it **must** use the temporary array. All of this is possible because during the code generation, we know if we are "before" or "after" the array assignment and the sink of the dependence.

Before explaining how to integrate, in a practical manner, the use of temporaries within a parallelizing algorithm, we show the following complexity result.

Theorem 1. *Given a MDG (in dimension 1), it is NP-complete to determine if it is possible, by introducing at most K arrays, to transform the MDG into a directed graph with no circuit (i.e., with maximal parallelism).*

Proof. By reduction from FEEDBACK-ARC-SET (problem GT8 in [5]). Starting from any directed graph G, we derive a code whose MDG is G, except with a few modifications that do not change circuits. Each statement is surrounded by an if that generates a path anti-dependence. When a vertex in G has an out-degree larger than 1, we add intermediate vertices and flow dependences so that all vertices have only one out-going path anti-dependence. The code below corresponds to the transformation of a graph G with 2 vertices a and b with 2 arcs from b to a (this is why we added the vertex c) and 1 arc from a to b. The cheapest (in terms of memory) is to break in G the arc from a to b which means to store the first condition. This is the only possibility with one temporary array.

```
do i = 2, n − 1
   if (b(i + 1) > 0) then
      a(i) = 1
   if (a(i + 1) > 0) then
      b(i) = 2
   if (a(i + 1) ≥ 0) then
      c(i) = b(i − 1) + 3
enddo
```

```
doall i = 2, n − 1
   t1(i) = (b(i + 1) > 0)
enddo
doall i = 2, n − 1
   if (a(i + 1) > 0) then b(i) = 2
enddo
doall i = 2, n − 1
   if (a(i + 1) ≥ 0) then c(i) = b(i − 1) + 3
enddo
doall i = 2, n − 1
   if (t1(i)) then a(i) = 1
enddo
```

3.3 Parallelizing Algorithm

We now explain how to integrate the use of temporary arrays into a parallelizing algorithm based on loop distribution such as Allen, Callahan, and Kennedy's algorithm [2]. Note first that, if no control structure is involved in an anti dependence, we can directly use the MDG without temporary arrays: the situation is the same as for static control codes. If we want to retrieve McKinley and Kennedy's approach, we can apply the transformation of Section 3.2 for all path anti-dependence, i.e., allow maximal dimensions for temporary arrays. For intermediate dimensions, we will parameterize Allen, Callahan, and Kennedy's algorithm by fixing, for each control structure involved in a path anti-dependence, the maximal dimension t we allow for a temporary array to store the condition: $t = -1$ if no temporary is allowed, and $-1 \leq t \leq d$ where d is the depth (number of do vertices along the control path in the CDG) of the expression to be stored.

We apply the same technique as Allen, Callahan, and Kennedy's algorithm: we start with $k = 1$ for generating the outermost loops, we compute the strongly connected components of the MDG, we determine a valid partition of vertices (see the conditions in Section 2), we order the groups following a topological order, we remove all dependences satisfied at level k (either loop carried, or

between two different groups), and we start again for level $k+1$. The only difference is that we determine, on the fly, if we do the transformation of a path anti-dependence: this is possible as soon as $k > d - t$ (and we say that the temporary array is *activated*). Indeed, t is the "amount" of expression that we can store in the available memory: the $d-t$ missing dimensions create an output dependence for the outermost loops that we "remove" by declaring temporary array as *privatized* for the outer loops that are parallel. For code generation, control expressions use either the original expression or the temporary array depending on their position with respect to the activated array, as we explained in Section 3.2. All details (and several illustrating examples) can be found in [10].

Remarks: (1) An interesting idea in McKinley and Kennedy's approach is to use a 3 state logic for avoiding "cascades" of conditionals. This technique can also be incorporated in our framework (see again [10]). (2) To find the minimal required dimensions, we can check all possible configurations for the parameters t, which seems feasible on real (portion of) codes. Indeed, in practice, the nesting depth is small and there are only a few control structures involved in **anti**-dependences, even for codes with many control structures.

4 Conclusion

In this paper, we presented a new type of graph to take into account data dependences involving if and do control structures. This graph allows us to use the classical algorithm of Allen, Callahan, and Kennedy even in the case of codes with nonstatic control flow. We also explained how we can control the dimensions of the temporary arrays that are introduced by adding parameters to the graph. More details on this work can be found in the PhD thesis of the second author [10]. The algorithm has been implemented in Nestor [11], a tool to implement source to source transformations of Fortran programs.

References

1. W. Abu-Sufah. *Improving the Performance of Virtual Memory Computes.* PhD thesis, Dept. of Comp. Science, University of Illinois at Urbana-Champaign, 1979.
2. J. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proc. of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, Germany, Jan. 1987.
3. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, Jan. 1983.
4. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.
6. K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing'90*, Aug. 1990.

7. Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Feb. 1971.
8. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec. 1986.
9. V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journ. of Research & Development*, 41(3):233–264, May 1997.
10. G.-A. Silber. *Parallélisation automatique par insertion de directives*. PhD thesis, École normale supérieure de Lyon, France, Dec. 1999.
11. G.-A. Silber and A. Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, vol. 1593 of *LNCS*, pages 653–662. Springer-Verlag, Apr. 1999.
12. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

Automatic Generation of Block-Recursive Codes*

Nawaaz Ahmed and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. Block-recursive codes for dense numerical linear algebra computations appear to be well-suited for execution on machines with deep memory hierarchies because they are effectively blocked for all levels of the hierarchy. In this paper, we describe compiler technology to translate iterative versions of a number of numerical kernels into block-recursive form. We also study the cache behavior and performance of these compiler generated block-recursive codes.

1 Introduction

Locality of reference is important for achieving good performance on modern computers with multiple levels of memory hierarchy. Traditionally, compilers have attempted to enhance locality of reference by *tiling* loop-nests for each level of the hierarchy [4, 10, 5]. In the dense numerical linear algebra community, there is growing interest in the use of block-recursive versions of numerical kernels such as matrix multiply and Cholesky factorization to address the same problem. Block-recursive algorithms partition the original problem recursively into problems with smaller working sets until a base problem size whose working set fits into the highest level of the memory hierarchy is reached. This recursion has the effect of blocking the data at many different levels at the same time. Experiments by Gustavson [8] and others have shown that these algorithms can perform better than tiled versions of these codes.

To understand the idea behind block-recursive algorithms, consider the iterative version of Cholesky factorization shown in Figure 1. It factorizes a symmetric positive definite matrix `A` into the product $A = L \cdot L^T$ where L is a lower triangular matrix, overwriting A with L. A block-recursive version of the algorithm can be obtained by sub-dividing the arrays A and L into 2×2 blocks, as shown in Figure 2. Here, $chol(X)$ computes the Cholesky factorization of array X. The recursive version factorizes the A_{00} block, performs a division on the A_{10} block, and finally factorizes the updated A_{11} block. The termination condition for the recursion can be either a single element of A (in which case a square root operation is performed) or a $b \times b$ block of A which is factored by the iterative code.

* This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 368–378, 2000.
© Springer-Verlag Berlin Heidelberg 2000

```
    for j = 1, n
        for k = 1, j-1
            for i = j, n
S1:             A(i,j) -= A(i,k) * A(j,k)
S2:     A(j,j) = dsqrt(A(j,j))
        for i = j+1, n
S3:         A(i,j) = A(i,j) / A(j,j)
```

Fig. 1. Cholesky Factorization

$$\begin{bmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} L_{00}^T & L_{10}^T \\ 0 & L_{11}^T \end{bmatrix}$$

$$= \begin{bmatrix} L_{00}L_{00}^T & L_{00}L_{10}^T \\ L_{10}L_{00}^T & L_{10}L_{10}^T + L_{11}L_{11}^T \end{bmatrix}$$

$$L_{00} = chol(A_{00})$$

$$L_{10} = A_{10}L_{00}^{-T}$$

$$L_{11} = chol(A_{11} - L_{10}L_{10}^T)$$

Fig. 2. Block recursive Cholesky

A block-recursive version of matrix multiplication $C = AB$ can also be derived in a similar manner. Subdividing the arrays into 2×2 blocks results in the following block matrix formulation —

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

Each matrix multiplication results in eight recursive matrix multiplications on sub-blocks. The natural order of traversal of a space in this recursive manner is called a *block-recursive* order and is shown in Figure 4 for a two-dimensional space. Since there are no dependences, the eight recursive calls to matrix multiplication can be performed in any order. Another way of ordering these calls is to make sure that one of the operands is reused between adjacent calls[1]. One such ordering corresponds to traversing the sub-blocks in the *gray-code* order. A gray-code order on the set of numbers $(1 \ldots m)$ arranges the numbers so that adjacent numbers differ by exactly 1 bit in their binary representation. A gray-code order of traversing a 2-dimensional space is shown in Figure 5. Such an order is called *space-filling* since the order traces a complete path through all the points, always moving from one point to an adjacent point. There are other space-filling orders; some of them are described in the references [6]. Note that lexicographic order, shown in Figure 3, is not a space-filling order.

In this paper, we describe compiler technology that can automatically convert iterative versions of array programs into their recursive versions. In these programs, arrays are referenced by affine functions of the loop-index variables. As a result, partitioning the iterations of a loop will result in the partitioning of data as well. We use affine mapping functions to map all the statement instances of the program to a space we call the *program iteration space*. This mapping effectively converts the program into a perfectly-nested loop-nest in which all statements are nested in the innermost loop. We develop legality conditions under which the iteration space can be recursively divided. Code is then generated to traverse the space in a block-recursive or space-filling manner, and when each point in this space is visited, the statements mapped to it are executed. This strategy effectively converts the iterative versions of codes into their recursive

[1] Not more than one can be reused, in any case.

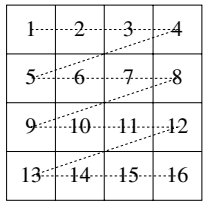


Fig. 3. Lexicographic

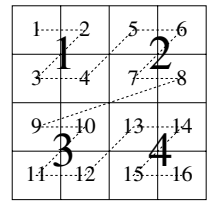


Fig. 4. Block Recursive

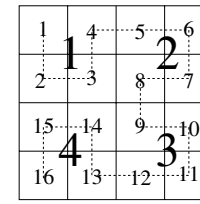


Fig. 5. Space-Filling

ones. The mapping functions that enable this conversion can be automatically derived when they exist. This approach does not require that the original program be in any specific form – any sequence of perfectly or imperfectly nested loops can be transformed in this way.

The rest of this paper is organized as follows. Section 2 gives an overview of our approach to program transformation (details are in [2]); in particular, in Section 2.1, we derive legality conditions for recursively traversing the program iteration space. Section 3 describes code generation, and Section 4 presents experimental results. Finally, Section 5 describes future work.

2 The Program-Space Formulation

A program consists of statements contained within loops. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use S_1, S_2, ..., S_n to name the statements of the program in syntactic order.

A *dynamic instance* of a statement S_k refers to a particular execution of that statement for a given value of index variables i_k of the loops surrounding it, and is represented by $S_k(i_k)$. The execution order of these instances can be represented by a *statement iteration space* of $|i_k|$ dimensions, where each dynamic instance $S_k(i_k)$ is mapped to the point i_k. For the iterative Cholesky code shown in Figure 1, the statement iteration spaces for the three statements `S1`, `S2` and `S3` are $j_1 \times k_1 \times i_1$, j_2, and $j_3 \times i_3$ respectively. The program execution order of a code fragment can be modeled in a similar manner by a *program iteration space*, defined as follows.

1. Let $\mathcal{P}$ be the Cartesian product of the individual statement iteration spaces of the statements in that program. The order in which this product is formed is the syntactic order in which the statements appear in the program. If p is the sum of the number of dimensions in all statement iteration spaces, then $\mathcal{P}$ is p-dimensional. $\mathcal{P}$ is also called the *product space* of the program.
2. Embed all statement iteration spaces $\mathcal{S}_k$ into $\mathcal{P}$ using embedding functions $\tilde{F}_k$ which satisfy the following constraints:
   (a) Each $\tilde{F}_k$ must be one-to-one.

(b) If the points in space $\mathcal{P}$ are traversed in lexicographic order, and all statement instances mapped to a point are executed in original program order when that point is visited, the program execution order is reproduced.

The program execution order can thus be modeled by the pair $(\mathcal{P}, \tilde{\mathcal{F}} = \{\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_n\})$. We will refer to the program execution order as the *original* execution order. For the Cholesky example, the program iteration space is a 6-dimensional space $\mathcal{P} = j_1 \times k_1 \times i_1 \times j_2 \times j_3 \times i_3$. One possible set of embedding functions $\tilde{\mathcal{F}}$ for this code is shown below –

$$\tilde{F}_1(\begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}) = \begin{bmatrix} j_1 \\ k_1 \\ i_1 \\ j_1 \\ j_1 \\ i_1 \end{bmatrix} \quad \tilde{F}_2(\begin{bmatrix} j_2 \end{bmatrix}) = \begin{bmatrix} j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \end{bmatrix} \quad \tilde{F}_3(\begin{bmatrix} j_3 \\ i_3 \end{bmatrix}) = \begin{bmatrix} j_3 \\ j_3 \\ i_3 \\ j_3 \\ j_3 \\ i_3 \end{bmatrix}$$

Note that all the six dimensions are not necessary for our Cholesky example. Examining the mappings shows us that the last three dimensions are redundant and the program iteration space could as well be 3-dimensional. For simplicity, we will drop the redundant dimensions when discussing the Cholesky example. The redundant dimensions can be eliminated in a systematic manner by retaining only those dimensions whose mappings are linearly independent.

In a similar manner, any other execution order of the program can be represented by an appropriate pair $(\mathcal{P}, \mathcal{F})$. Code for executing the program in this new order can be generated as follows. We traverse the entire product space lexicographically, and at each point of $\mathcal{P}$ we execute the original program with all statements protected by guards. These guards ensure that only statement instances mapped to the current point are executed. For the Cholesky example, naive code which implements the execution order $(\mathcal{P}, \tilde{\mathcal{F}})$[2] is shown in Figure 6. This naive code can be optimized by using standard polyhedral techniques [9] to remove the redundant loops and to find the bounds of loops which are not redundant. An optimized version of the code is shown in Figure 7. The conditionals in the innermost loop can be removed by index-set splitting the outer loops.

2.1 Traversing the Program Iteration Space

Not all execution orders $(\mathcal{P}, \mathcal{F})$ respect the semantics of the original program. A legal execution order must respect the *dependences* present in the original program. A dependence is said to exist from instance i_s of statement S_s to instance i_d of statement S_d if both statement instances reference the same array location, at least one of them writes to that location, and instance i_s occurs before instance i_d in original execution order. Since we traverse the product space lexicographically, we require that the vector $v = F_d(i_d) - F_s(i_s)$ be lexicographically positive for every pair (i_s, i_d) between which a dependence exists. We refer to v as the *difference vector*.

[2] We have dropped the last three redundant dimensions for clarity.

```
for j1 = -inf to +inf
for k1 = -inf to +inf
for i1 = -inf to +inf
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:       A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:     A(i,j) = A(i,j) / A(j,j)
```

Fig. 6. Naive code for Cholesky

```
for j1 = 1,n
for k1 = 1,j1
for i1 = j1,n
    if (k1 < j1)
S1:    A(i1,j1) -= A(i1,k1) * A(j1,k1)

    if (k1==j1 && i1==j1)
S2:    A(j,j) = dsqrt(A(j1,j1))

    if (k1==j1 && i1 > j1)
S3:    A(i1,j1) = A(i1,1j) / A(j1,j1)
```

Fig. 7. Optimized code for Cholesky

For a given embedding $\mathcal{F}$, there may be many legal traversal orders of the product space other than lexicographic order. The following traversal orders are important in practice.

1. Any order of walking the product space represented by a unimodular transformation matrix T is legal if $T \cdot v$ is lexicographically positive for every difference vector v associated with the code.
2. If the entries of all difference vectors corresponding to a set of dimensions of the product space are **non-negative**, then those dimensions can be blocked. This partitions the product space into blocks with planes parallel to the axes of the dimensions. These blocks are visited in lexicographic order. This order of traversal for a two-dimensional product space divided into equal-sized blocks is shown in Figure 3.
   When a particular block is visited, all points within that block can be visited in lexicographic order. Other possibilities exist. Any set of dimensions that can be blocked can be *recursively* blocked. If we choose to block the program iteration space by bisecting blocks recursively, we obtain the block-recursive order shown in Figure 4.
3. If the entries of all difference vectors corresponding to a dimension of the product space are **zero**, then that dimension can be traversed in any order. If a set of dimensions exhibit this property, then those dimensions can not only be blocked, but the blocks themselves do not have to be visited in a lexicographic order. In particular, these blocks can be traversed in a *space-filling* order. This principle can be applied recursively within each block, to obtain space-filling orders of traversing the entire sub-space (Figure 5).

Given an execution order $(\mathcal{P}, \mathcal{F})$, and the dependences in the program, it is easy to check if the difference vectors exhibit the above properties using standard dependence analysis [11]. If we limit our embedding functions $\mathcal{F}$ to be affine functions of the loop-index variables and symbolic constants, we can determine functions that allow us to block dimensions (and hence also recursively block them) or to traverse a set of dimensions in a space-filling order. The condition

that entries corresponding to a particular dimension of all difference vectors must be non-negative (for recursive-blocking) or zero (for space-filling orders) can be converted into a system of linear inequalities on the unknown coefficients of $\mathcal{F}$ by an application of *Farkas' Lemma* as discussed in [2]. If this system has solutions, then any solution satisfying the linear inequalities would give the required embedding functions. The embedding functions for the Cholesky example were determined by this technology.

3 Code Generation

Consider an execution order of a program represented by the pair $(\mathcal{P}, \mathcal{F})$. We wish to block the program iteration space recursively, terminating when blocks of size $B \times B \ldots \times B$ are reached.

Let p represent the number of dimensions in the product space. To keep the presentation simple, we assume that redundant dimensions have been removed and that all dimensions can be blocked. We also assume that all points in the program iteration space that have statement instances mapped to them are positive and that they are contained in the bounding box $(1 \cdots B \times 2^{k_1}, \ldots, 1 \cdots B \times 2^{k_p})$.

Code to traverse the product space recursively is shown in Figure 8. The parameter to the procedure `Recurse` is the current block to be traversed, its co-ordinates given by `(lb[1]:ub[1], ..., lb[p]:ub[p])`. The function `HasPoints` prevents the code from recursing into blocks that have no statement instances mapped to them. If there are points in the block and the block is not a base block, `GenerateRecursiveCalls` subdivides the block into 2^p sub-blocks by bisecting each dimension and calls `Recurse` recursively in a lexicographic order[3]. The parameter `q` of the procedure `GenerateRecursiveCalls` specifies the dimension to be bisected. On the other hand, if the parameter to `Recurse` is a base block, code for that block of the iteration space is executed in procedure `BaseBlockCode`.

For the initial call to `Recurse`, the lower and upper bounds are set to the bounding box.

Naive code for `BaseBlockCode(lb,ub)` is similar to the naive code for executing the entire program. Instead of traversing the entire product space, we only need to traverse the points in the current block lexicographically, and execute statement instances mapped to them. Redundant loops and conditionals can be hoisted out by employing polyhedral techniques.

Blocks which contain points with statement instances mapped to them can be identified by creating a linear system of inequalities with variables lb_i, ub_i corresponding to each entry of `lb[1..p]`, `ub[1..p]` and variables x_i corresponding to each dimension of the product-space. Constraints are added to ensure that the point $(x_1, x_2, \ldots, x_p)$ has a statement instance mapped to it and that it lies within the block $(lb_1 : ub_1, \ldots, lb_p : ub_p)$. From the above system, we obtain the condition to be tested in `HasPoints(lb,ub)` by projecting out (in the Fourier-Motzkin sense) the variables x_i.

[3] This must be changed appropriately if space-filling orders are required

```
Recurse(lb[1..p], ub[1..p])
if (HasPoints(lb,ub)) then
   if (∀i ub[i] == lb[i]+B-1) then
      BaseBlockCode(lb)
   else
      GenerateRecursiveCalls(lb,ub,1)
   endif
endif
end


GenerateRecursiveCalls(lb[1..p], ub[1..p], q)
if (q > p)
   Recurse(lb, ub)
else
  for i = 1,p
     lb'[i] = lb[i]
     ub'[i] = (i==q) ? (lb[i]+ub[i])/2
                     : ub[i]
  GenerateRecursiveCalls(lb',ub',q+1)

  for i = 1, p
     lb'[i] = (i==q) ? (lb[i]+ub[i])/2 + 1
                     : lb[i]
     ub'[i] = ub[i]
  GenerateRecursiveCalls(lb',ub',q+1)
endif
end
```

Fig. 8. Recursive code generation

```
BaseBlockCode(lb[1..3])
for j1 = lb[1], lb[1]+B-1
for k1 = lb[2], lb[2]+B-1
for i1 = lb[3], lb[3]+B-1
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:       A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:     A(i,j) = A(i,j) / A(j,j)
end


HasPoints(lb[1..3], ub[1..3])
if (lb[1]<=n && lb[2]<=n && lb[3]<=n
    && lb[1]<=ub[3]
    && lb[2]<=ub[1]
    && lb[2]<=ub[3])
    return true
else
    return false
end
```

Fig. 9. Recursive code for Cholesky

For our Cholesky example, the embedding functions shown in Section 2 enable all dimensions to be blocked. Since there are difference vectors with non-zero entries, the program iteration space cannot be walked in a space-filling manner, though it can be recursively blocked. The portion of the product-space that has statement instances mapped to it is $[j, k, i] : 1 \leq k \leq j \leq i \leq n$. This is used to obtain the condition in `HasPoints()`. Naive code for executing the code in each block is shown in Figure 9. As mentioned earlier, the redundant loops must be removed and the conditionals hoisted out for good performance.

4 Experimental Results

In this section, we discuss the performance of block-recursive and space-filling codes produced using the technology described in this paper. All experiments were run on an SGI R12K machine running at 300Mhz with a 32Kb primary-data cache (L1), 2Mb second-level cache (L2) and 64 TLB entries.

The legality conditions discussed in Section 2.1 permit us to conclude matrix multiply (MMM) can be blocked both recursively and in a space-filling manner. The Cholesky code can only be blocked recursively. We generated four versions of block-recursive code with different base block sizes (16, 32, 64, 128) for both programs. These codes were compiled with the "-O3 -LNO:blocking=off" option

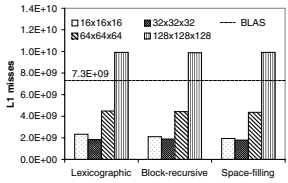


Fig. 10. MMM : L1 cache

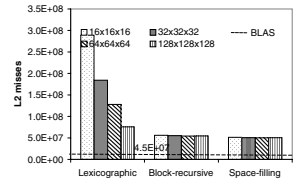


Fig. 11. MMM : L2 cache

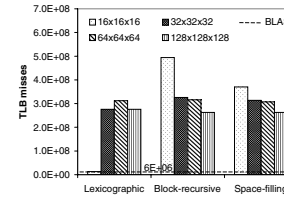


Fig. 12. MMM : TLB

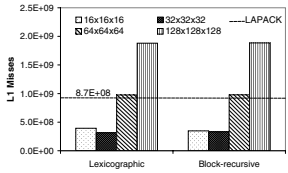


Fig. 13. CHOL : L1 cache

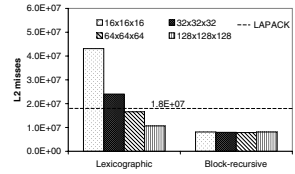


Fig. 14. CHOL : L2 cache

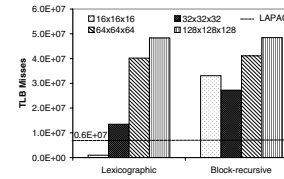


Fig. 15. CHOL : TLB

of the SGI compiler. At this level of optimization, the SGI compiler performs tiling for registers and software-pipelining.

For each program, we ran the recursive (and if legal, the space-filling) versions of the code for a variety of matrix sizes. For lack of space, we only present results for a matrix size of 4000×4000. Results for other matrix sizes are similar. In the graphs, the results marked `Lexicographic` correspond to executing the code in `BaseBlockCode(lb)` by visiting the base blocks in a lexicographic order. For comparison, we also show results of executing vendor-supplied, hand-tuned implementations of matrix multiply (`BLAS`) and Cholesky (`LAPACK` [3]).

Figures 10 and 13 show the number of L1 data cache misses for the two programs. For the larger block sizes (64, 128), the data touched by a base-block does not fit in cache (32K) and hence both the recursive and lexicographic versions suffer the same penalty. For smaller block sizes (16, 32), the data does fit into cache resulting in much fewer misses. Figures 11 and 14 show the L2 cache misses. The lexicographic versions for block sizes of 16 and 32 exhibit much higher miss numbers than the corresponding recursive versions since these block sizes are too small to fully utilize the 2M cache. In the recursive versions, however, even the small block sizes succeed in full utilization of the cache due to the recursive doubling effect. These recursive versions will have a similar effect on any further levels of caches. Of the two recursive orders, the space-filling orders show slightly better cache performance for both programs.

Figures 12 and 15 show the number of TLB misses for the two programs. The R12K TLB has only 64 entries, hence large block sizes (more than 64) will exhibit high miss rates in both the lexicographic and recursive cases. Small block sizes could work well in the lexicographic case if the loop order is chosen well. In our case, the `jki` order is the best order for both the programs. There are very few TLB misses when the block size is 16 because fewer than 48 TLB entries are required at a time for this block size. In the recursive case, the recursive

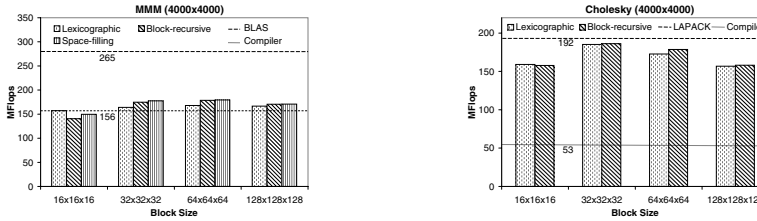


Fig. 16. Performance

doubling does cause significantly more TLB misses for small block sizes, although the recursive walks are largely immune to the effect of reordering the loops. By comparison, in the `jik`-order (not shown here), the code with block size of 16 suffers a 100-fold increase in the number of TLB misses for the lexicographic case but the number of misses remains roughly the same in the recursive cases.

Figure 16 shows the performance of the two programs in MFlops. As a sanity check, the lines marked `Compiler` show the performance obtained with compiling the original code with the "-O3" flag of the SGI compiler which attempts to tile for cache and registers and then software-pipeline the resulting code. For both programs, the recursive codes with block size of 32 are the best among all the generated code. For most block sizes, the recursive codes are better than their lexicographic counter-parts by a small percentage (2-5%). For a block size of 16, the recursive cases are worse due to an increase in the number of TLB misses.

For matrix multiply, the best recursive code generated by the compiler is still substantially worse than the hand-tuned versions of the programs even though the recursive overhead is less than 1% in all cases. This difference could be due to the high number of TLB misses suffered by the recursive versions. Copying data from base blocks into contiguous locations as is done in the hand-tuned code might help improve performance. It is interesting to note that although the hand-tuned version suffers higher primary cache miss rates, the impact on performance is small. This is not surprising in an out-of-order issue processor like the R12K where the latency of primary cache misses (10 cycles) can be hidden by scheduling and software-pipelining. These misses will be more important in an in-order issue processor like the Merced. For Cholesky factorization, on the other hand, the best block-recursive version is comparable in performance to LAPACK code.

5 Related Work and Conclusions

Hand-coded versions of block recursive algorithms have been studied for a long time [1, 7, 6]; some of them are implemented in the IBM's Engineering and Scientific Subroutine Library (ESSL) for example.

In this paper, we developed program restructuring technology to convert iterative numerical programs into block-recursive versions. Our experiments show

that the block-recursive versions of matrix multiply and Cholesky are effectively blocked for all memory hierarchy levels. However, base block sizes must be chosen with care – the data accessed in a base-block must fit into the lowest level of the cache hierarchy, the blocks must be large enough so that the recursive overhead is negligible, and the back-end compiler must be able to schedule the instructions in a base-block efficiently. Unfortunately, our experiments also show that the block-recursive algorithms do not interact well with the TLB. In spite of this, the best compiler-generated code for the two applications was nevertheless a recursive version. We conjecture that better interaction with the TLB requires either (i) copying data from column-major order into recursive data layouts as suggested by Chatterjee [6] or (ii) copying the data used by a base block into contiguous locations as suggested by Gustavson [1].

The work in this paper can be extended in a number of ways. More experiments are needed to assess the importance of block-recursive codes for other applications such as relaxation methods. Non-square base-blocks may be useful to eliminate conflict misses in some codes. Finally, it would be interesting to study the effect of copying data into layouts that are matched to block-recursive traversals.

References

[1] Ramesh C. Agarwal, Fred G. Gustavson, Joan McComb, and Stanley Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.

[2] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[4] Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.

[5] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[6] S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99)*, June 1999.

[7] Matteo Frigo, C.L.Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science*. IEEE Press, 1999.

[8] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[9] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.

[10] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.

[11] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, August 1992.

Left-Looking to Right-Looking and Vice Versa: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring*

Nikolay Mateev, Vijay Menon, and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. We have recently developed a new program analysis strategy called *fractal symbolic analysis* that addresses some of limitations of techniques such as dependence analysis. In this paper, we show how fractal symbolic analysis can be used to convert between left-looking and right-looking versions of three kernels of central importance in computational science: Cholesky factorization, LU factorization with pivoting, and triangular solve.

1 Introduction

Most computational science codes require the solution of linear systems of equations. These systems can be written as $Ax = b$ where A is a matrix, b is a vector of known values, and x is the vector of unknowns. Direct methods for solving linear systems factorize the matrix A into the product of an upper triangular matrix and a lower triangular matrix, and then find x by solving the two triangular systems. If the matrix is symmetric and positive-definite, Cholesky factorization is usually used to find the two triangular factors; otherwise, LU with partial pivoting is used.

Substantial effort has been invested by the numerical analysis community in implementing high-performance versions of these algorithms. For example, the LAPACK library contains blocked implementations of these algorithms, optimized to perform well on a memory hierarchy [2]; the SCALAPACK library contains parallel implementations of these algorithms for distributed-memory machines [3].

In the compiler community, researchers have developed techniques to synthesize blocked and parallel implementations of these algorithms from high-level algorithmic formulations. These *restructuring* techniques perform source-to-source transformations to improve parallelism and locality of reference. A significant challenge for compiler optimization is the fact that there are many variations in how these algorithms can be expressed. The two most important variations are called *right-looking* or *eager*, and *left-looking* or *lazy*.

* This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 379–388, 2000.
© Springer-Verlag Berlin Heidelberg 2000

- *Eager*: In matrix factorization codes, the matrix is walked by column from left to right. After the current column has been computed, updates to columns to the right of the current column are performed immediately. Similarly in triangular solves, the current unknown is computed, and its contribution is immediately subtracted from the remaining equations. In the numerical analysis community, these are referred to as *right-looking* formulations.
- *Lazy*: Updates to the current element/column from earlier elements/columns are performed as late as possible, in a lazy manner. These are also referred to as *left-looking* formulations.

The effectiveness of different compiler optimizations can be sensitive to the original formulation. The storage layout of a matrix in memory or across multiple processors may lead to a preference for one or the other of these formulations. Thus, it is important for compilers to transform one form to the other.

The most commonly used technique for proving legality of transformations is *dependence analysis* [8], which computes and enforces a partial order between the statements based upon data dependences. A more powerful technique that subsumes dependence analysis is *symbolic analysis*, which compares symbolically two programs for equality. Both approaches have their shortcomings. The constraints imposed by dependence analysis are sufficient but not necessary, and fail to prove equality of right- and left-looking LU. Symbolic analysis, on the other hand, is precise but intractable for all but the simplest programs.

To bridge this gap between dependence analysis and symbolic analysis, we developed *fractal symbolic analysis* [6]. In this paper, we show how this new analysis technique can be used to convert between left- and right-looking versions of triangular solve, Cholesky factorization, and LU factorization with partial pivoting. In Section 2, we abstract the transformation required to convert between left- and right-looking formulations, and show that dependence analysis is too weak to prove the equality of left- and right-looking versions of LU factorization with pivoting. In Section 3, we summarize fractal symbolic analysis. In Section 4, we demonstrate its effectiveness in verifying the legality of these transformations on LU with pivoting (for lack of space, we do not discuss triangular solve and Cholesky, but both dependence analysis and fractal symbolic analysis are adequate for these programs. These and other details can be found in an expanded version of this paper [7]). Finally, we conclude with future directions.

2 Factorizations and Triangular Solve

In this section, we discuss right- and left-looking formulations of three important numerical kernels: Cholesky factorization, LU factorization with pivoting, and lower triangular solve. Neither right- nor left-looking forms should be viewed as canonical in general. For example, in [4], a standard text on matrix computations, Cholesky and lower triangular solve are introduced in a left-looking or lazy manner, while LU is introduced in a right-looking or eager manner. It should

```
do k = 1,n
  B1(k);
  do j = k+1,n
    B2(k,j);
```

(a) Eager/Right-looking Code

```
do j = 1,n
  do k = 1,j-1
    B2(k,j);
  B1(j);
```

(b) Lazy/Left-looking Code

Fig. 1. Equivalent Right-looking and Left-looking Codes

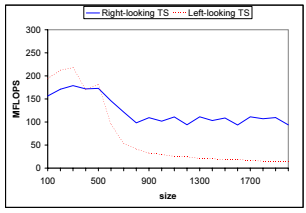


Fig. 2. Triangular Solve

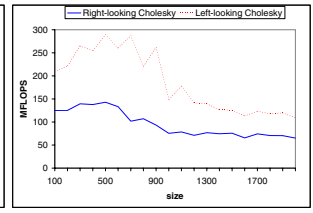


Fig. 3. Cholesky

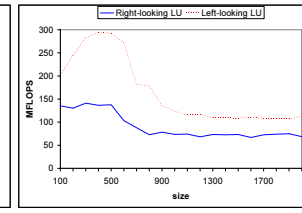


Fig. 4. LU with Pivoting

be noted that this has no correlation with performance. To illustrate this, we present the performance of both forms on an SGI Octane[1].

At a high-level, the transformation between left- and right-looking versions can be viewed as a transformation we call *right-left loop interchange*, illustrated in Figure 1. In each of the codes discussed below, right-looking formulations correspond to Figure 1a, and left-looking formulations correspond to Figure 1b. The underlying operations, denoted by `B1` and `B2`, are the same in both cases. We show that conversion between right- and left-looking formulations may be accomplished by one or more applications of right-left loop interchange.

2.1 Lower Triangular Solve

Triangular solve, shown in Figure 5, maps directly to the template of Figure 1. Both `B1` and `B2` are represented by a single statement. `B1` corresponds to the final scaling step of solving a single equation with one unknown, and `B2` corresponds to the substitution of a solved unknown (`x(k)`) to compute an unsolved unknown (`x(j)`). Although triangular solve is often introduced in its left-looking form (as in [4]), the right-looking form can sometimes be desirable for performance. When compiled in Fortran on the SGI Octane, the right-looking form considerably outperforms the left-looking form as shown in Figure 2. Here, the right-looking code has better spatial locality as `A` is stored in column-major order.

2.2 Cholesky Factorization

Our second example is Cholesky factorization, a key computational kernel for factoring symmetric, positive-definite matrices. Figure 6 presents both right- and

[1] This 300MHz machine has a 2MB L2 cache, and an R12K processor. All compiled code was generated using the SGI MIPSpro f77 compiler with flags: -O3 -n32 -mips4.

```
     do k = 1,n
       // Compute current unknown
B1(k) : x(k) = x(k)/A(k,k)

       // Update from current unknown
       // to later unknowns
       do j = k+1,n
B2(k,j) : x(j) = x(j)-A(j,k)*x(k)
```

(a) Right-looking Triangular Solve

```
     do j = 1,n
       // Update from earlier unknowns
       // to current unknown
       do k = 1,j-1
B2(k,j) : x(j) = x(j)-A(j,k)*x(k)

       // Compute current unknown
B1(j) : x(j) = x(j)/A(j,j)
```

(b) Left-looking Triangular Solve

Fig. 5. Lower Triangular Solve

```
     do k = 1,N
       // Scale current column
B1(k) : A(k,k) = sqrt(A(k,k))
       do i = k+1,N
         A(i,k) = A(i,k)/A(k,k)

       // Update from current column
       // to columns to right
       do j = k+1,N
B2(k,j) : do i = j,N
            A(i,j) = A(i,j)-A(i,k)*A(j,k)
```

(a) Right-looking Cholesky

```
     do j = 1,N
       // Update from columns
       // to left to current column
       do k = 1,j-1
B2(k,j) : do i = j,N
            A(i,j) = A(i,j)-A(i,k)*A(j,k)

       // Scale current column
B1(j) : A(j,j) = sqrt(A(j,j))
       do i = j+1,N
         A(i,j) = A(i,j)/A(j,j)
```

(b) Left-looking Cholesky

Fig. 6. Cholesky Factorization

left-looking versions of this operation. As in the case of triangular solve, Cholesky maps directly to the template suggested in Figure 1. In this case, `B1` and `B2` are represented by small blocks of code. `B1` corresponds to the computation in the current column, and `B2` corresponds to updates from earlier columns to the left to later columns to the right. As before, performance is sensitive to the formulation that is used. However, in this case, as shown in Figure 3, the left-looking formulation results in better performance.

2.3 LU Factorization with Partial Pivoting

Our last example is LU factorization with partial pivoting, which is used for factoring general unsymmetric matrices. Without pivoting, LU factorization is quite similar to Cholesky in the previous section, but suffers from instability due to accumulating floating point error. In practice, partial pivoting provides a solution to this problem.

For this example, converting between a right-looking formulation (as in Figure 7a) and a left-looking formulation (as in Figure 7b) is a more involved process. The pivot operation performed in each column requires a corresponding swap of elements in every other column. This swap can be viewed as a second 'update' between columns. Conversion between right- and left forms may be accomplished by two applications of right-left loop interchange. In the right-looking code in Figure 7a, the update alone may be converted to left-looking form, as in Figure 7c. Converting the swap is slightly more complicated, as the swap is

```
do k = 1, N
  // Pick the pivot
B1.a(k) :
  p(k) = k
B1.b(k) :
  do i = k+1, N
    if abs(A(i,k)) > abs(A(p(k),k))
      p(k) = i

  // Swap rows
B1.c(k) :
  do j = 1, N
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Scale current column
B1.d(k) :
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k)

  // Update from current column
  // to columns to right
  do j = k+1, N
B2(k,j) :
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
```

(a) Right-looking LU

```
do j = 1, N
  // Swap rows from left
  do k = 1, j-1
    tmp = A(k,j)
    A(k,j) = A(p(k),j)
    A(p(k),j) = tmp

  // Update from columns to left
  // to current column
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
```

(b) Left-looking LU

```
do j = 1, N
  // Update to current column
  // from columns to left
  do k = 1, j-1
B2(k,j) :
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
B1.a(j) :
  p(j) = j
B1.b(j) :
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows
B1.c(j) :
  do k = 1, N
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
B1.d(j) :
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)
```

(c) Hybrid Right-Left LU #1

```
do j = 1, N
  // Update to current column
  // from columns to left
  do k = 1, j-1
    do i = k+1, N
      A(i,j) = A(i,j) - A(i,k)*A(k,j)

  // Pick the pivot
  p(j) = j
  do i = j+1, N
    if abs(A(i,j)) > abs(A(p(j),j))
      p(j) = i

  // Swap rows to the left
  do k = 1, j
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp

  // Scale current column
  do i = j+1, N
    A(i,j) = A(i,j) / A(j,j)

  // Swap rows to the right
  do k = j+1, N
    tmp = A(j,k)
    A(j,k) = A(p(j),k)
    A(p(j),k) = tmp
```

(d) Hybrid Right-Left LU #2

Fig. 7. LU Factorization with Partial Pivoting

```
        do i = 1,N
          do j = 1,N
S(i,j) :    k = k + A(i,j)
```

Fig. 8. Reduction

$$A(\boldsymbol{k}) = \begin{cases} guard_1(\boldsymbol{k}) \rightarrow expression_1(\boldsymbol{k}) \\ guard_2(\boldsymbol{k}) \rightarrow expression_2(\boldsymbol{k}) \\ \qquad\qquad \vdots \\ guard_n(\boldsymbol{k}) \rightarrow expression_n(\boldsymbol{k}) \end{cases}$$

Fig. 9. Guarded Symbolic Expression

never purely right-looking since pivoting requires swaps in earlier columns as well as latter columns. Nevertheless, the right-looking portion of the swap may be isolated, via index-set-splitting and statement reordering [8], as in Figure 7d. A second application of right-left loop interchange produces the left looking code in Figure 7b.

Figure 4 shows the performance of these codes on the SGI Octane. Although the right-looking version is simpler, the left-looking version has notably better cache performance. One key obstacle to automatic conversion between right- and left-looking forms is the inability of dependence analysis to establish their equivalence. At the level of matrix operations, the pivot swaps may be viewed as row permutations and the updates as matrix multiplications. In converting between right- and left-looking forms, these operations are interchanged, but they are not independent since they modify certain common storage locations. Hence, a compiler that relies on dependence analysis will not be able to prove the equivalence of these versions. In the next section, we present a more powerful analysis tool that can establish the legality of this transformation.

3 Fractal Symbolic Analysis

In this section, we give a brief overview of *fractal symbolic analysis*, a technique we proposed in [6] to establish legality of program transformations. As mentioned earlier, dependence analysis is too conservative to handle a code such as LU factorization with pivoting. Traditional symbolic analysis, on the other hand, is generally impractical. Fractal symbolic analysis provides an accurate and tractable means of analyzing many codes.

To illustrate the basic idea, consider the simple example in Figure 8. For a number of reasons, a compiler may desire to interchange the `i` and `j` loops in this code. However, every instance of the statement `S` writes to the variable `k`. As a result, dependence analysis enforces a total ordering between all instances of `S` and, therefore forbids loop interchange. Nevertheless, if commutativity and associativity of addition is allowed, this interchange produces equivalent results. Most modern compilers would use pattern recognition to figure out that the interchange is legal. However, pattern recognition is notoriously fragile, so a more robust test is desirable. Symbolic analysis is one option, but direct symbolic comparison of programs is usually intractable.

Transformation	Legality Condition

Loop Interchange
`do i = 1,n`
`  do j = 1,m`
`  S(i,j);`
`<->`
`do j = 1,m`
`  do i = 1,n`
`  S(i,j);`
	$commute(\langle S(p,q), S(r,s)\rangle :$
	$1 <= p < r <= n \wedge 1 <= s < q <= m)$

Right/Left-looking Interchange
`do k = 1,n`
`  S1(k);`
`  do j = k+1,n`
`  S2(k,j);`
`<->`
`do j = 1,n`
`  do k = 1,j-1`
`  S2(k,j);`
`  S1(j);`
	$commute(\langle S1(t), S2(r,s)\rangle :$
	$1 <= r < t < s <= n) \wedge$
	$commute(\langle S2(p,q), S2(r,s)\rangle :$
	$1 <= p < r < s < q <= n)$

Fig. 10. Legality Conditions for Program Transformations

Commute Condition	Recursive Condition
Statement Sequence $commute(\langle$ `S1; S2;...; SN,B2` $\rangle : cond)$	$commute(\langle S1, B2\rangle : cond) \wedge$ $commute(\langle S2, B2\rangle : cond) \wedge$ ... $commute(\langle SN, B2\rangle : cond)$
Loop $commute(\langle$ `do i = l,u` `  S1(i);` `,B2` $\rangle : cond)$	$commute(\langle S1(i), B2\rangle : cond \wedge l <= i <= u)$

Conditional Statement
$commute(\langle$ `if (pred) then`
`  S1;`
`else`
`  S2;`
`,B2` $\rangle : cond)$
	$commute(\langle S1, B2\rangle : cond \wedge pred) \wedge$
	$commute(\langle S2, B2\rangle : cond \wedge \neg pred)$

Fig. 11. Recursive Simplification Rules

The key idea behind fractal symbolic analysis is the following. Loop interchange reorders particular instances of statements. This reordering may be viewed *incrementally* by interchanging instances *one pair at a time*. In the example above, the legality of loop interchange is established by symbolically demonstrating that two individual instances, `S(i,j)` and `S(i',j')`, commute (that is, that they can be done in any order). This only requires proving that k_{out} `=` k_{in} `+ A(i,j) + A(i',j')` and k_{out} `=` k_{in} `+ A(i',j') + A(i,j)` are equivalent, which may be verified by a relatively simple symbolic engine.

In general, there are two aspects to fractal symbolic analysis: (i) recursive simplification, and (ii) base symbolic comparison.

3.1 Recursive Simplification

As discussed above, fractal symbolic analysis simplifies programs recursively till they are simple enough for the base symbolic comparison engine. There are three key ideas to this simplification. First, if the programs to be compared are too complex for symbolic comparison, simplified programs are generated such that equality of the simplified programs is a sufficient, but not in general necessary condition to establish the equality of the original codes. Second, for codes obtained by common program transformations, the appropriate simplification may

be derived from the transformation as in the example above. Figure 10 provides the legality conditions for both the loop interchange performed above and the right-left loop interchange presented earlier in this paper. Finally, this simplification process may be applied recursively until tractable programs are obtained. Figure 11 provides rules for recursive simplification.

3.2 Base Symbolic Comparison

Although compared programs may be repeatedly simplified as needed, each simplification step results in a loss of accuracy as equality of the simplified programs is a sufficient but not necessary condition. Because of this, it is important to symbolically compare programs with as few simplification steps as possible. In [6], we describe a core symbolic comparison engine that is effective under the following constraints. Recursive simplification may be applied until these constraints are met.

- Programs consist of assignment statements, for-loops and conditionals.
- Loops do not carry dependences.
- Array indices and loop bounds are restricted to be affine functions of enclosing loop variables and symbolic constants, and predicates are restricted to be conjunctions and disjunctions of affine inequalities.

Under these conditions, we have shown that the effect of a program on each live, modified variable may be summarized as a *guarded symbolic expression*, as shown in Figure 9. Each guard describes a polyhedral region of array indices, and the corresponding expression describes the values of the array elements for those indices. Computation and comparison of guarded symbolic expressions is a straightforward process and is described in detail in [6].

4 LU with Pivoting

We now demonstrate how fractal symbolic analysis can be applied to establish the legality of the right-left transformation on LU factorization with pivoting. For conciseness, we will focus on the equivalence of the codes in Figure 7a and 7c. As discussed earlier, these codes differ by a single application of right-left loop interchange. Since reordered operations are not independent, dependence analysis is insufficient to establish legality. On the other hand, our implementation of fractal symbolic analysis, described in the last section and in greater detail in [6], is able to automatically verify the legality of this transformation. In this section, we describe this process.

Recall that dependence analysis cannot prove Figure 7a and 7c equivalent due to dependences between reordered swaps ($B1.c$) and updates ($B2$). However, given the fact that $k \leq p(k)$,[2] the two codes still produce the same results.

[2] The predicate $k \leq p(k)$ is easily inferred from the code using techniques such as array value propagation [5].

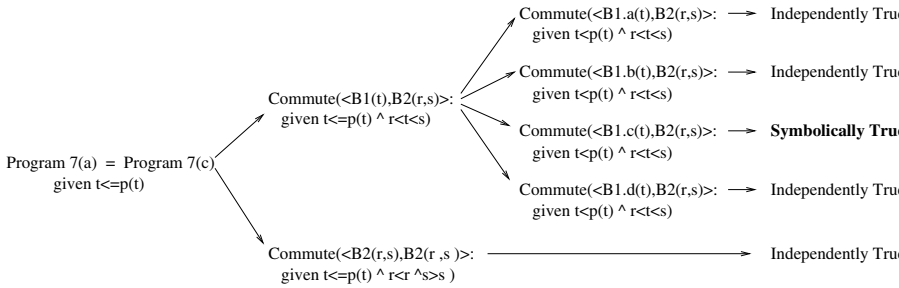


Fig. 12. Fractal Symbolic Analysis of LU

$B2(m,n)$:

```
do i = m+1, N
  A(i,n) = A(i,n) - A(i,m)*A(m,n)
```

$B1.c(l)$:

```
do k = 1, N
  tmp = A(l,k)
  A(l,k) = A(p(l),k)
  A(p(l),k) = tmp
```

(a) $B2(m,n); B1.c(l)$

$B1.c(l)$:

```
do k = 1, N
  tmp = A(l,k)
  A(l,k) = A(p(l),k)
  A(p(l),k) = tmp
```

$B2(m,n)$:

```
do i = m+1, N
  A(i,n) = A(i,n) - A(i,m)*A(m,n)
```

(b) $B1.c(l); B2(m,n)$

Fig. 13. Simplified Comparison

Fractal symbolic analysis deduces correctly that these codes are equal. Figure 12 illustrates this process on the two codes. Essentially, fractal symbolic analysis is able to reduce the legality of the right-left interchange to the symbolic legality of reordering swaps ($B1.c$) and updates ($B2$).

This simpler legality test is illustrated in in Figure 13. Dependences are still violated, but these programs are "simple enough" to be compared by direct symbolic analysis. The only live, altered variable in either program is the array `A`, and the core symbolic engine generates equivalent guarded symbolic expressions for `A` from each program:

$$A_{out}(i,j) = \begin{cases} i = l \wedge j = n & \rightarrow A_{in}(p(l),n) - A_{in}(p(l),m) * A_{in}(m,n) \\ i = p(l) \wedge j = n & \rightarrow A_{in}(l,n) - A_{in}(l,m) * A_{in}(m,n) \\ i = l \wedge j \neq n & \rightarrow A_{in}(p(l),j) \\ i = p(l) \wedge j \neq n & \rightarrow A_{in}(l,j) \\ i \neq l \wedge i \neq p(l) \wedge j = n & \rightarrow A_{in}(i,n) - A_{in}(i,m) * A_{in}(m,n) \\ i \neq l \wedge i \neq p(l) \wedge j \neq n & \rightarrow A_{in}(i,j) \end{cases}$$

At this point, the symbolic expressions corresponding to each guard are syntactically equivalent. This need not be the case in general. However, in this case, it demonstrates that the programs in Figure 13 (and, thus, the original codes in Figure 7a and 7c) are computationally equivalent. That is, fractal symbolic

analysis is able to demonstrate that no floating point computation is reordered between right- and left-looking formulations of LU factorization with partial pivoting, therefore the transformation does not affect numerical stability.

5 Conclusions

In this paper, we have studied right and left formulations for three important linear algebra kernels and argued the importance of automatically converting between the two formulations. Furthermore, we have abstracted the high-level transformation that equates the two formulations of these codes. We have discussed how fractal symbolic analysis may be used to establish the legality of this transformation, and have demonstrated its applicability to LU factorization with pivoting, a case in which dependence analysis fails. As far as we are aware, fractal symbolic analysis is the only technique general enough to equate left and right formulations for all the examples mentioned in this paper.

As a future goal, we would like to *synthesize* transformation sequences using fractal symbolic analysis. Dependence information can be represented abstractly using dependence vectors or polyhedra, and these representations have been exploited to synthesize transformation sequences [1, 8]. At present, we do not know suitable representations for the results of fractal symbolic analysis, nor do we know how to synthesize transformation sequences from such information.

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proc. International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[3] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, 1997.

[4] Gene Golub and Charles Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996.

[5] V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *Proc. International Conference on Supercomputing*, pages 265–269, July 1995.

[6] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis for program transformations. Technical Report TR2000-1781, Cornell University, Computer Science, January 2000.

[7] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Left-looking to right-looking and vice versa: An application of fractal symbolic analysis to linear algebra code restructuring. Technical Report TR2000-1797, Cornell University, Computer Science, June 2000.

[8] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1995.

Identifying and Validating Irregular Mutual Exclusion Synchronization in Explicitly Parallel Programs

Diego Novillo[1], Ronald C. Unrau[1], and Jonathan Schaeffer[2]

[1] Red Hat Inc., Sunnyvale, CA 94089, USA
{dnovillo,runrau}@redhat.com
[2] Computing Science Department, University of Alberta, Edmonton, Alberta, Canada T6G 2H1
jonathan@cs.ualberta.ca

Abstract. Existing work on mutual exclusion synchronization is based on a structural definition of mutex bodies. Although correct, this structural notion fails to identify many important locking patterns present in some programs. In this paper we present a novel analysis technique for identifying mutual exclusion synchronization patterns in explicitly parallel programs. We use this analysis in a new technique, called *lock-picking*, which detects and eliminates redundant mutex operations. We also show that this new mutex analysis technique can be used as a validation tool in a compiler. Using this analysis, a compiler can detect irregularities like lock tripping, deadlock patterns, incomplete mutex bodies, dangling `lock` and `unlock` operations and partially protected code.

1 Introduction

In this paper we present a novel analysis technique for identifying mutual exclusion synchronization patterns in explicitly parallel programs. We apply this analysis to develop a new technique, called *lock-picking*, to detect and eliminate redundant mutex operations. We also show that this new mutex analysis technique can be used as a validation tool in a compiler. We build on a concurrent data-flow analysis framework called CSSAME (Concurrent Static Single Assignment with Mutual Exclusion,pronounced *sesame*) [6] to analyze and optimize the synchronization framework of both task and data parallel programs. We have implemented these algorithms and apply them to several concurrent and sequential applications.

2 The CSSAME Form

The CSSAME form is a refinement of the Concurrent SSA (CSSA) framework [3] that incorporates mutual exclusion synchronization analysis to identify memory interleavings that are not possible at runtime due to the synchronization structure of the program. CSSAME extends CSSA to include mutual exclusion synchronization and barrier synchronization [5].

Like the sequential SSA form, CSSAME has the property that every use of a variable is reached by exactly one definition. Two merge operators are used in the CSSAME

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 389–394, 2000.
© Springer-Verlag Berlin Heidelberg 2000

form: ϕ functions and π functions. A ϕ function merges all the incoming control reaching definitions to create a new definition for the variable. Control reaching definitions are those that reach a use u via sequential flow of execution (i.e., the definition has been made by the same thread). The second merge operator is the π function, which merges concurrent reaching definitions. Concurrent reaching definitions are those that reach a use u from other threads.

3 Motivation and Overview

Given an arbitrary statement s in a program and a lock variable L, a mutex structure analyzer should be able to answer the question "does s execute under the protection of lock L?". The answer to that question should be one of *always*, *never* or *sometimes*. To be conservatively correct, the compiler treats *never* and *sometimes* as equivalent. Furthermore, if the analysis determines that statement s is sometimes protected and sometimes not, this information could be used to warn the user about an anomalous locking pattern.

Existing work on mutual exclusion synchronization is based on a structural definition of mutex bodies [2, 4, 6]. A mutex body is indicated by a pair of `lock` and `unlock` nodes. All the graph nodes dominated by the `lock` node and post-dominated by the `unlock` node are part of the mutex body. Although correct, this notion of mutex body fails to identify some valid locking patterns present in some programs.

For example, consider the code fragment in Figure 1, which is part of a quicksort algorithm taken from the TreadMarks DSM system. We are interested in the mutual exclusion sections created by the lock variable *TSL*. Notice that a structural definition of mutex bodies will identify no mutex bodies in this function. The only `lock`/`unlock` pair that might qualify as a mutex body are the statements L_1 and U_3 (lines 3 and 37 respectively). However, the presence of other `lock` and `unlock` operations in between these statements forces the compiler to disregard this pair as a valid mutex body. A closer inspection reveals that the only statement that executes without lock protection is the busy wait statement S_1 (line 24).

4 Detecting Mutex Structures

A mutex structure for lock variable L is the set of all the mutex bodies for L in the program. To detect mutex structures, the intermediate representation for the program is modified so that (a) every graph node contains a use for *each* lock variable in the program, and, (b) for each lock variable L the graph entry node is assumed to contain an `unlock(L)` operation (i.e., variables are initially "unlocked").

Mutex structures are detected using sequential reaching definition information for each lock variable L. Nodes that are only reached by definitions of L coming from `lock(L)` nodes are protected by L. Nodes that can be reached by at least one `unlock(L)` node are not protected by L. Using this information we build an initial set of mutex bodies for each individual `lock(L)` node in the graph. This initial set is then

```
1  int PopWork(TaskElement *task)
2  {
3    L1 ⇒ lock(TSL);
4    while (TaskStackTop == 0) {
5      if (++NumWaiting == NPROCS) {
6        /* All the threads are waiting for work.
7         * We are done.
8         */
9        lock(pause_lock);
10       pause_flag = 1;
11       unlock(pause_lock);
12       U1 ⇒ unlock(TSL);
13       return DONE;
14     } else {
15       if (NumWaiting == 1) {
16         lock(pause_lock);
17         pause_flag = 0;
18         unlock(pause_lock);
19       }
20       U2 ⇒ unlock(TSL);
21       /* Wait for work. This is the only
22        * statement not protected by TSL.
23        */
24       S1 ⇒ while (!pause_flag) ; /* busy-wait */
25       L2 ⇒ lock(TSL);
26       if (NumWaiting == NPROCS) {
27         U3 ⇒ unlock(TSL);
28         return DONE;
29       }
30       --NumWaiting;
31     } /* endif ++NumWorking == NPROCS */
32   } /* while task-stack empty */
33   /* Pop a piece of work from the stack */
34   TaskStackTop--;
35   task->left  = TaskStack[TaskStackTop].left;
36   task->right = TaskStack[TaskStackTop].right;
37   U3 ⇒ unlock(TSL);
38   return 0;
39 }
```

Fig. 1. Locking pattern in function *PopWork()*.

refined by merging mutex bodies with common nodes [5]. This mutex analysis framework can be used as a validation tool in a compiler. Using this analysis, a compiler can detect irregularities like [5]:

Lock Tripping. Let L be a lock variable and n be a `lock(L)` node. Suppose that n is reached by other `lock(L)` nodes. If all the definitions come from other `lock(L)` nodes, the program is guaranteed to trip over lock L at runtime. If only some definitions come from other `lock(L)` nodes, the program may or may not trip over lock L.

Deadlock. Let L and M be two different lock variables such that in thread T_1 there is a `lock(L)` node that reaches a `lock(M)` node. In another thread T_2 a `lock(M)` node reaches a `lock(L)` node. If both T_1 and T_2 can execute concurrently, then the program may deadlock at runtime.

Incomplete mutex bodies. Let $B_L(n)$ be a partially built mutex body for L such that no node in $B_L(n)$ is an `unlock(L)` node. At runtime, if lock L is acquired at n, it will not be released.

Dangling unlock operations. Let x be an unlock node for L such that the set of reaching definitions for L at x does not include a `lock(L)` node. This indicates that the calling thread is releasing a lock that it has not acquired.

5 Lock-Picking

Sometimes it is possible to remove synchronization operations from a program without affecting its semantics. For example, mutual exclusion synchronization is unnecessary in a sequential program and can be safely removed. In this section we describe *lock-picking*, a transformation that finds and removes superfluous `lock` and `unlock` operations. We say that a mutex body can be *lock-picked* if its lock and unlock nodes can be

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S3 = π(S0, S1, S2);
    R3 = π(R0, R1, R2);
    lock(R1);
    for (j = 0; j < M; j++) {
      sum_reduction(A[i][j]);
    }
    unlock(R2);
  }
  ...
}

sum_reduction(double x)
{
  S4 = π(S0, S1, S2)
  R4 = π(R0, R1, R2)
  lock(S1);
  Sum = Sum + x;
  unlock(S2);
}
```

(a) Original CSSAME form.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    S3 = π(S0, S1, S2)
    R3 = π(R0, R1, R2)
    lock(R1);
    for (j = 0; j < M; j++) {
      S4 = π(S0, S1, S2)
      lock(S1);
      Sum = Sum + A[i][j];
      unlock(S2);
    }
    unlock(R2);
  }
  ...
}
```

(b) CSSAME form after inlining and π pruning.

```
double Sum = 0;
parloop (p, 0, N) {
  ...
  for (i = 0; i < M; i++) {
    R3 = π(R0, R1, R2)
    lock(R1);
    for (j = 0; j < M; j++) {
      Sum = Sum + A[i][j];
    }
    unlock(R2);
  }
  ...
}
```

(c) After lock-picking.

Fig. 2. Effects of lock-picking on nested mutex bodies.

removed. An important property of lock picking is that it *does not* need to examine the mutex bodies of the program. Only the lock and unlock nodes are analyzed.

The lock-picking algorithm [5] examines the lock nodes for every mutex body in the program. The decision to lock-pick a mutex body is based on the absence of π functions for one or more lock variables at each mutex body lock node. The absence of π functions for lock variables at lock nodes means that there are no concurrent threads trying to acquire that lock. These conditions are typically discovered using whole program analysis. For example, consider the program in Figure 2(a). The inner loop calls the function *sum_reduction* to update a global reduction variable. Since *sum_reduction* is a generic reduction function, it locks the variable before doing the reduction. However, as a result of inlining, reduction lock S is no longer necessary because the reduction is always protected by lock R (Figure 2(b)). When *sum_reduction* is inlined, the use of R at the lock node for S becomes a protected use and its π function can be removed [6] (Figure 2(c)). In this case we say that the mutex structure for lock S is *nested* inside the mutex structure for R.

6 Experimental Results

We selected programs originally written in Java because we anticipated optimization opportunities due to the thread-safe nature of its libraries. Since Java libraries are thread-safe, application programs may spend up to half their execution time performing

Benchmark	Unoptimized time (secs)	Optimized time (secs)	Relative Speedup
Array (1,000)	23	20	1.15
Array (10,000)	547	534	1.02
Map (3,000)	32	30	1.07
Map (30,000)	273	227	1.20
Sort (3,000)	32	30	1.07
Sort (30,000)	407	327	1.24

Table 1. Effect of lock-picking (LP) on sequential Java programs.

unnecessary synchronization [1]. The key reason for this overhead is that the libraries are generic and are not specific to an individual application's context. Hence, they have to be conservative in the assumptions they make. Therefore, when considered within the context of an actual program it might turn out that most of the synchronization operations are not necessary.

Table 1 shows the improvements obtained by applying lock-picking to sequential Java programs found in the JGL abstract class library these programs. We executed both the Java and C versions of these programs; in both cases the results were similar. In general, we obtained performance improvements between 2% and 24% when lock-picking was applied. The performance gains obtained by removing the unnecessary locks are directly related to this particular implementation of mutual exclusion. Since these are sequential programs, all the synchronization overhead is caused by the actual call to `lock` and `unlock`. There is no lock contention. An alternative to removing the locks would have been to use a more efficient mutual exclusion synchronization implementation. We are convinced that a combination of compiler optimizations and efficient lock implementations is the best approach in these cases.

7 Conclusions

Synchronization analysis techniques are important in the context of an optimizing compiler for explicitly parallel programs. By reducing the number of memory conflicts, they simplify subsequent analysis and allow more aggressive optimizations to be applied.

In this paper we have developed a new technique to analyze non-concurrency for mutex synchronization that can handle locking patterns not supported by existing techniques. This allows the analysis of more complex mutual exclusion synchronization patterns in explicitly parallel programs. We have shown that this analysis can help detect common locking irregularities in parallel programs. Finally, we apply this analysis to remove mutex synchronization when it can be proven superfluous.

References

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN '98*, June 1998.

[2] A. Krishnamurthy and K. Yelick. Analyses and Optimizations for Shared Address Space Programs. *J. Parallel and Distributed Computing*, 38:130–144, 1996.
[3] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *LCPC '97*, August 1997.
[4] S. P. Masticola. *Static Detection of Deadlocks in Polynomial Time*. PhD thesis, Department of Computer Science, Rutgers University, 1993.
[5] D. Novillo. *Compiler Analysis and Optimization Techniques for Explicitly Parallel Programs*. PhD thesis, University of Alberta, February 2000.
[6] D. Novillo, R. Unrau, and J. Schaeffer. Concurrent SSA Form in the Presence of Mutual Exclusion. In *ICPP '98*, pages 356–364, August 1998.

Exact Distributed Invalidation

Rupert W. Ford[1], Michael F.P. O'Boyle[2], and Elena A. Stöhr[1]

[1] Department of Computer Science, The University, Manchester M13 9PL, U.K.
[2] Department of Computer Science, The University of Edinburgh, Mayfield Rd., Edinburgh EH9 3JZ, U.K.

Abstract. This paper develops and proves an exact distributed invalidation algorithm for programs with compile time decidable control-flow. We present an efficient constructive algorithm that globally combines locally gathered information to insert coherence calls in such a manner that eliminates all invalidation traffic without loss of locality and places the minimal number of coherence calls. Experimental results show that it outperforms existing compiler directed coherence techniques and hardware based memory consistency.

1 Introduction

The main goal of any distributed shared memory system is to support a shared memory programming model across distributed resources as efficiently as possible. More specifically, we would like to minimise the system overhead associated in maintaining memory coherence. This paper focuses on reducing the amount of coherence traffic associated with maintaining consistency. In particular we are interested in entirely eliminating invalidation traffic in a write-invalidation based protocol using compiler directed distributed invalidation. Given certain preconditions, we can provably eliminate all invalidation traffic thereby reducing latency. Furthermore, we can easily expand this approach to tackle the general case without adversely increasing memory traffic.

In invalidation protocols, attempts to write a new value may be delayed until all remote copies are invalidated. Performance can be degraded both by the delay on the writing node, and by the resulting network traffic. One approach to improving performance is to reduce the overhead of invalidation traffic within a write-invalidate based protocol by using distributed invalidation (DI). DI [10] transfers the responsibility of invalidation from the writing processor to the processors with the remote copies. The writing processor does not incur a write miss and can proceed without stalling as the invalidation of copies is done locally. The DI scheme also has the advantage of reducing network invalidation traffic by removing invalidation and acknowledgement messages.

Early work invalidated all cached data at each parallel region or epoch. More recent schemes have used tags or timestamps to maintain cached data across epochs [2,3,5]. Some schemes use a compiler controlled directory to help in runtime dependence analysis, whilst others remove the need for a directory altogether [2,11,6,12,15,17]. In [4], a more sophisticated form of analysis based on

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 395–404, 2000.
© Springer-Verlag Berlin Heidelberg 2000

reads after RDS (relaxed determining sequence) is described. In [7], vectorisation is used to minimise the overhead of redundant invalidation when using RDS analysis. Array data-flow analysis was used in [5] to detect and eliminate stale data references. However, the greater accuracy of this method was not exploited in determining the coherence action of the entire program.

In our previous work [13] we presented an algorithm based on coherence equations which allowed the exact modelling of coherence actions. However, this approach, in general, is undecidable and relies on a heuristic technique. This paper describes an exact algorithm for distributed invalidation for static control-flow programs and makes the following contributions:

- it presents an exact compiler algorithm that provably eliminates all invalidation traffic in static control-flow programs.
- it eliminates unnecessary invalidations which cause loss of temporal locality and provably places the minimal number of self-invalidations to maintain consistency.

2 Approach

In DI if each processor invalidates its stale read copies and marks as exclusive the data it will write then memory consistency is maintained without incurring any invalidation traffic by inserting local invalidate calls (LI) and local exclusive calls (LEx) (see [13]). The goal of any compiler directed technique is to determine exactly those memory elements that require coherence actions. Under estimation will partly rely on the systems coherence mechanism, over estimation leads to over invalidation and loss of temporal locality.

1. Forall statements, determine enclosing static if conditions
2. For each loop nest L
   (a) For each loop with the nest deepest first *(section 5)*
      i. Determine coherence actions required
      ii. Insert coherence code
      iii. Determine exposed writes and live reads
      iv. Remove anti-dependences and consider each loop as a statement
3. For a basic block *(section 4)*
   (a) Determine coherence actions required
   (b) Insert coherence code

Fig. 1. The coherence algorithm

Given a certain sequence of memory accesses, we can exactly determine the coherence actions required, based on array section analysis and static scheduling information. The key feature of our algorithm is that relatively short sequences can be summarised locally before combining the results to determine the coherence actions throughout the program. In figure 1, starting at the lowest loop

nest depth, the statements are examined to see if there exists a cross-processor anti-dependence. If there is a dependence, only those read and write actions occurring within the loop at that level are considered. Once coherence calls have been inserted, it is necessary to determine those upwardly exposed writes that may form the sink of anti-dependences causing coherence traffic. Similarly we need to determine those reads that are not covered by coherence calls and may be the source of anti-dependence causing later coherence traffic. The loop nest can now be considered as a single statement with a modified read and write set. This approach not only guarantees that all anti-dependences are exactly determined, it also means that coherence calls are always placed at the highest lexical level - removing the overhead of repeatedly making redundant coherence calls.

2.1 Example

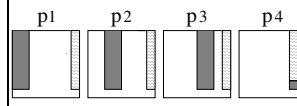
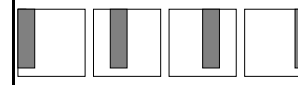
Column 1 of figure 2 shows a parallelised program fragment based loosely on the Eispack routine Tred2, where `lo` and `hi` refer to the local upper and lower loop bounds. Column 2 shows a series of 4 boxes denoting the particular memory actions at various stages in the program to array z, in each of the four processors, while column 3 presents a summary of the state of memory at various points - critical to our algorithm. In column 2, reading or writing data that is already in exclusive state on the local processor does not affect it's memory state and is denoted by the colour grey. Similarly, reads to local data already in read-only state remain in read-only state denoted by a box containing a wave pattern. Reading remote data requires data to be in read state, once again denoted by a wave pattern. When a processor wishes to write data previously in read state, it must first mark the data to be written exclusive, i.e. Ro->Ex, with a call to LEx, this is denoted by the area of memory marked in black. Remote read copies must also be invalidated, i.e. Ro->I, with a call to LI, denoted by the cross [1].

We first of all consider those deepest nested loops containing statements S2 and S3 and only consider those read and write actions within the immediate enclosing loop. As these are parallel loops there are no cross-processor anti-dependences and we may simply summarise the read and write actions as an array section for later use. Moving up syntactic levels, we must also consider statement S1 then S4 and cross-processor dependences within L1.

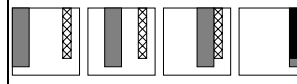
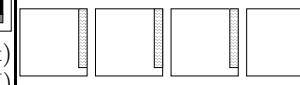
In the case of the write in statement S3, the cross-processor anti-dependence from S2 to S3 is from one iteration to the next. Hence, in S3 the read copies invalidated corresponding to those of the previous iteration. Once the coherence actions have been determined within loop L1, they must be summarised before the whole fragment must be considered. In particular we must consider those read copies that may still be sources of anti-dependences and those write accesses that may be sinks. If we examine column 3, we notice that there are read copies of the final column after executing L1. This uncovered read has a cross-processor dependence with S4, hence the coherence calls in the final entry of column 2.

[1] Coherence calls are only shown for one case due to space restrictions.

Parallel Program	Reads/Writes/ Coherence Actions	Exposed Writes and Reads

`if (n.ne.1) then`
`L1:Do i=2,n`
`if (lo<=i-1<=hi)then`
`S1: z(i-1,i-1)=1.0`
`endif`
`Do j=lo,min(hi,i-1)`
`Do k=1,i-1`
`S2: gg(j)=gg(j)+z(k,i)*z(k,j)`
`Enddo`
`Enddo`
	
	S2,i=n: Read Non-Local (Ex->Ro) (wave), Read Local (Ex) (grey), Read Local (Ro) (wave).
		
		L1: Exposed write, assumed to be Ex.

`<Barrier+Coherence code>`		

`Do j=lo,min(hi,i-1)`
`Do k=1,i-1`
`S3: z(k,j)=z(k,j)-gg(j)*d(k)`
`Enddo`
`Enddo`
`Enddo`
	
	S3,i=n: just Write (Ex) (grey), Invalidate (Ro->I) (X), Mark Ex. and Write (Ro->Ex) (black).
		
		L1: Uncovered reads in Ro.

`call mp_barrier()`
`if (lo<=n<=hi) then`
`LEx(z(1,n))`
`else`
`LI(z(1,n))`
`endif`
`endif`		

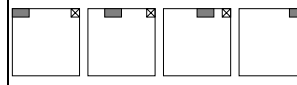
`Do i=lo,hi`
`S4: z(1,i)=0.0`
`Enddo`
	
	S4: just Write (Ex) (grey), Invalidate (Ro->I) (X), Mark Ex. and Write (Ro->Ex) (black).	

Fig. 2. Illustrative Example

3 Coherence Equations

This section summarises equations describing coherence based on [13]. Let an action i be a read or write operation occurring after action $i-1$ and before $i+1$. Let W_i and R_i be the set of all the pages (global data) written and read respectively in action i and let $\underline{W}_i$ and $\underline{R}_i$ be the corresponding local pages. Let $\underline{\mathcal{E}x}_i$ and $\underline{\mathcal{R}o}_i$ be the set of local pages in exclusive and read state respectively after action i. A superscript is used, if necessary, to distinguish between actions occurring on different processors, e.g. $\underline{W}_i^1$ and $\underline{W}_i^2$ refer to the local data written on processors 1 and 2 respectively.

We use an owner-computes rule and static scheduling. To eliminate multiple writer false sharing, arrays are padded and partitioned along page boundaries where necessary [12].

After a write, the local pages in exclusive/read state will be modified as follows:

$$\underline{\mathcal{E}x}_i = \underline{\mathcal{E}x}_{i-1} \cup \underline{W}_i \tag{1}$$

$$\underline{\mathcal{R}o}_i = \underline{\mathcal{R}o}_{i-1} - (\underline{\mathcal{R}o}_{i-1} \cap W_i). \tag{2}$$

Let p be the number of processors and let z be the processor id of the local processor. After a read action, the local pages in read and exclusive state will be modified as follows:

$$\underline{\mathcal{R}o}_i^z = \underline{\mathcal{R}o}_{i-1}^z \cup ((\bigcup_{k \in \{1,\ldots,p\}}^{k \neq z} \underline{R}_i^k) \cap \underline{\mathcal{E}x}_{i-1}^z) \cup (\underline{R}_i^z - \underline{\mathcal{E}x}_{i-1}^z) = \underline{\mathcal{R}o}_{i-1}^z \cup \hat{R}_i^z \tag{3}$$

$$\underline{\mathcal{E}x}_i^z = \underline{\mathcal{E}x}_{i-1}^z - (\underline{\mathcal{E}x}_{i-1}^z \cap (\bigcup_{k \in \{1,\ldots p\}}^{k \neq z} \underline{R}_i^k)). \tag{4}$$

Let $\underline{LEx}_i$ and $\underline{LI}_i$ be the local pages to be set to exclusive and invalid state respectively due to action i. The local pages to be made exclusive are those which will be written locally and are currently in read state:

$$\underline{LEx}_i = \underline{W}_i \cap \underline{\mathcal{R}o}_{i-1}. \tag{5}$$

The local pages to be invalidated are those formerly in read state if written to by remote processors:

$$\underline{LI}_i = (W_i - \underline{W}_i) \cap \underline{\mathcal{R}o}_{i-1}. \tag{6}$$

The above equations (5) and (6), if honoured by the compiler, will eliminate invalidation traffic and unnecessary misses [2].

[2] apart from those due to read/write false-sharing [14].

3.1 Compiler Implementation

Static control-flow is a well defined form of program containing statements, loops, procedures and if statements, where we restrict the form of conditionals to affine functions of compile-time known values and constant run-time parameters. We apply if-conversion throughout the program with the generated guards modelled as additional constraints. Those pages to be invalidated have been defined in terms of set algebra. To be amenable to compiler analysis, they have to be expressed as array sections. In our implementation we make use of Omega Library [9] for set manipulation.

4 Basic Blocks

Although the equations in section 3 precisely define those array elements to mark exclusive/invalid, they are in general, undecidable, as they are recursively defined in terms of previous states. We derive a non-recursive formulation of the coherence equations for basic blocks which can be used as the basis for a constructive algorithm and translation to Presburger formulae.

A basic block program or fragment of code B is an ordered set of statements and memory actions. Adjacent read actions within a statement in B can be merged into one read action. Therefore, a general basic block program or code fragment B can be represented as $B = \{R_1, W_1, ..., R_n, W_n\}$.

We make the initial assumption that all previous actions prior to entering the basic block have already been dealt with, i.e. $\underline{\mathcal{R}o}_0 = \emptyset$.

Given the restraint on $\underline{\mathcal{R}o}_o$ and our restriction to basic blocks we may recursively enumerate equation (3) to a point $s-1$ and rearrange the resulting equation using set manipulation to get:

$$\underline{\mathcal{R}o}_{s-1} = \bigcup_{t=1}^{s-1} (\hat{\underline{R}}_t - \bigcup_{u=t}^{s-1} W_t) \cup \hat{\underline{R}}_s. \tag{7}$$

The value of $\underline{\mathcal{R}o}$ is then substituted in equations (5) and (6).

Theorem 1. *Equations (7), (5) and (6) exactly define the coherence actions required before a write statement W_s [8].*

Due to space constraints the proofs of all theorems have been omitted and can be found in [8].

Based on the above formulation we have the following efficient algorithm to insert coherence code in basic blocks:

1. Find the first sink of cross processor anti-dependence S_k.
2. Find the union of local cross processor anti-dependence reads $\hat{\underline{D}}_t$ at each source of cross processor anti-dependence with a sink is S_k:
   $\underline{\mathcal{D}}_{k-1} = \cup_{t=1}^{k-1} \hat{\underline{D}}_t$.

3. Determine which coherence units should be made exclusive and which should be invalidated before S_k:
   $\underline{LEx}_k^z = \underline{W}_k \cap (\bigcup_{k \in \{1,\ldots,p\}}^{k/\neq} \underline{\mathcal{D}}_{k-1}^z)$
   $\underline{LIn}_k^z = (W_k - \underline{W}_k) \cap \underline{\mathcal{D}}_{k-1}^z$.
4. Insert coherence calls between statements S_{k-1} and S_k.
5. Reduce the local cross processor anti-dependence reads at each source of cross processor anti-dependence with a sink in S_k by what has been written:
   $\underline{\hat{D}}_t = \underline{\hat{D}}_t - W_k$.
6. Delete all anti-dependences with sinks in S_k and repeat steps 1-6 till the end of the basic block is reached.

Theorem 2. *The Basic Block algorithm eliminates all invalidation coherence and guarantees memory consistency [8].*

Theorem 3. *The algorithm inserts the minimum number of invalidation calls [8].*

5 Loops

Consider a loop with $2n$ statements and N iterations. Let $\underline{\hat{R}}_t(i)$ be a read access within iteration i in statement t which causes a cross-processor anti-dependence. Similarly, $\underline{W}_t(i)$ denotes a local write at iteration i within statement t. Before we can express the equations denoting the read state, we first introduce a term Q to allow a more succinct presentation of the read state equation:

$$Q(j,i,t,s) = \underline{\hat{R}}_t(j) - \bigcup_{u=t}^{n} W_u(j) - \bigcup_{k=j+1}^{i-1} \bigcup_{t=1}^{n} W_t(k) - \bigcup_{u=1}^{s-1} W_u(i), \tag{8}$$

if $j < i$.

This equation summarises all the reaching reads from statement S_t in iteration j to statement S_s at iteration i. It takes into consideration all the intervening writes which reduce the amount of data in read state. In those cases where we are considering statements within the same iteration, we can simplify Q as follows:

$$Q(i,i,t,s) = \underline{\hat{R}}_t(i) - \bigcup_{u=t}^{s-1} W_u(i),$$

if $i = j$, $t < s$. Finally, $Q(i,i,s,s) = \underline{\hat{R}}_s(i)$. We now have two cases:

No cross iteration dependences: This case is very similar to that of the basic block - except that we have an additional parameter - namely the iterator. This can be expressed as follows:

$$\underline{\mathcal{R}o}_s(i) = \bigcup_{t=1}^{s} Q(i,i,t,s).$$

General Case: For the general case where there may be cross-iteration data dependence, we have the following expression:

$$\underline{\mathcal{R}o}_s(i) = \bigcup_{j=1}^{i-1} \bigcup_{t=1}^{n} Q(j,i,t,s) \bigcup_{t=1}^{s} Q(i,i,t,s). \tag{9}$$

Theorem 4. *Equations (9), (5) and (6) exactly determine the necessary coherence actions in a loop before a write in a statement $W_s(i)$ [8].*

5.1 Nested Loops and Summarising

Once coherence actions have been determined for this loop level, we determine the array section of those writes that are upwardly exposed to possible anti-dependences at a higher loop level or earlier statement, i.e. $\bigcup_{i=1}^{N} \bigcup_{s=1}^{n} (\underline{W}_s(i) - \underline{LEx}_s(i) \cup \underline{LI}_s(i))$.

Similarly the read set associated will be those reads that may form the sources of anti-dependences is simply the read state after the last iteration, i.e.$\underline{\mathcal{R}o}_n(N)$. This leads to the overall algorithm described in figure 1.

6 Experiments

We prototyped the above algorithm in our compiler MARS [1] and applied it to two benchmarks, `Power`, an iterative eigenvalue solver and `Cholsky`, a routine from the Spec92 benchmarks. Our scheme was compared with hardware based sequential consistency (SC) which is guaranteed not to over-invalidate data and a compiler based scheme which uses lazy distributed invalidation based on relaxed determined sequences and time stamps [4]. The resulting programs were run on our DELTA simulator and execution times and memory statistics were gathered and presented below.

In the `power` program, both RDS and DI are capable of entirely eliminating the invalidation traffic required by sequential consistency shown by (Total remote cache line invalidates). This was achieved in the DI case by simply inserting local invalidates (Total local invalidates), however, additional write-backs were also needed by the RDS scheme. As we assume that write-backs are relatively cheap and the cost of invalidation traffic relatively small, both schemes give a modest improvement over sequential consistency. For larger systems, the improvement would be more dramatic. We further assume that special time-reads and the checking of the entire cache for data to flush, required by RDS, have no runtime cost. In practice, this could be a significant overhead and RDS would perform much worse than the DI scheme.

In the `cholsky` program, both sequential consistency and DI give the same performance as there are no runtime cross-processor dependences. In the RDS case, however, conservative compiler analysis has inserted excessive invalidation calls and write-backs leading to extremely poor execution time.

In both cases, DI has the best execution time.

Power (SC)	1	2	4	8	16	32
Cycles (*10^3)	283,240	143,363	74,625	42,224	28,172	23,536
Total wait for invalidate (*10^3)	0	108	3,358	8,779	23,319	54,976
Total remote cache line invalidates	0	1,568	4,704	10,976	23,520	48,608
Write backs to main memory	0	0	0	0	0	0
Power (DI)						
Cycles (*10^3)	283,510	142,838	73,779	41,038	26,570	21,341
Total wait for invalidates	0	0	0	0	0	0
Total remote cache line invalidates	0	0	0	0	0	0
Total Local invalidates	0	1,568	4,704	10,976	23,520	48,608
Write backs to main memory	0	0	0	0	0	0
Power (RDS)						
Cycles (*10^3)	283,666	142,916	73,818	41,058	26,580	21,346
Cache Flush checks	18	18	18	18	18	18
Time-reads (*10^3)	50	100	201	401	803	1,606
Total remote cache line invalidates	0	0	0	0	0	0
Total local invalidates	3,008	4,544	7,616	13,760	26,048	50,624
Write backs	3,136	3,136	3,136	3,136	3,136	3,136

Cholsky (SC)	1	2	4	8	16	32
Cycles (*10^3)	29,189	14,617	7,332	3,691	1,874	975
Total wait for invalidates (*10^3)	0	0	0	0	0	0
Total remote cache line invalidates	0	0	0	0	0	0
Write backs to main memory	0	0	0	0	0	0
Cholsky (DI)						
Cycles (*10^3)	29,189	14,617	7,332	3,691	1,874	975
Total wait for invalidates	0	0	0	0	0	0
Total remote cache line invalidates	0	0	0	0	0	0
Total Local invalidates	0	0	0	0	0	0
Write backs to main memory	0	0	0	0	0	0
Cholsky (RDS)						
Cycles (*10^3)	30,304	38,852	99,010	120,731	153,203	187,934
Cache Flush checks	64	64	64	64	64	64
Time-reads (*10^3)	381	381	381	381	381	381
Total remote cache line invalidates	0	0	0	0	0	0
Total local invalidates	26,194	28,381	27,943	27,110	26,235	34,422
Write backs to main memory	70,912	70,912	70,912	70,912	70,912	70,912

7 Conclusion

This paper has presented, for the first time, an exact compiler based distributed invalidation algorithm. Assuming a static control-flow program we provably insert the minimal number of coherence calls to guarantee consistency, eliminating all coherence traffic and reducing network contention without destroying temporal re-use due to over-invalidation. Furthermore, we can outperform existing

hardware and compiler-based techniques. Future work will combine this exact technique with our previous work on general control-flow, based on a hybrid coherence based scheme.

References

1. Bodin F., O'Boyle M.F.P., **A Compiler Strategy for SVM**, Proc. of Workshop on Lang., Compilers and Runtime Sys. for Scalable Comp., May 1995.
2. Cheong H., Veidenbaum A.V., **Compiler Directed Cache Management in Multiprocessors**, IEEE Computer, 23(6):39-48, June 1990.
3. Cheong H., **Life-Span Strategy - A Compiler-Based Approach to Cache Coherence**, Proc. of Int. Conf. on Supercomp., July 1992.
4. Choi L., Yew P-C., **A Compiler-Directed Cache Coherence Scheme with Improves Intertask Locality**, Proc. of Supercomp.'94, Nov. 1994.
5. Choi L., Yew P-C., **Compiler analysis for cache coherence: Interprocedural array data-flow analysis and its impacts on cache performance**, Tech. Report, University of Illinois, Sep. 1996.
6. Darnell E., Kennedy K., **Cache Coherence Using Local Knowledge**, Proc. of Supercomp.'93, Nov. 1993.
7. Darnell E., Mellor-Crumney J.M., Kennedy K., **Automatic Software Cache Coherence through Vectorisation**, Proc. of Int. Conf. on SuperComp., July 1992.
8. Ford R.W., O'Boyle M.F.P., Stöhr E.A., **Exact Distributed Invalidation**, Tech. Report, Dept. of Computer Science, Univ. of Manchester, 2000.
9. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T, and Wonnacott D., **The Omega Library Interface Guide**, Tech. Report, Dept. of Computer Science, Univ. of Maryland, 1996.
10. Lebeck A.R., Wood D.A., **Dynamic Self Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors**, Proc. of Inter. Symp. on Comp. Arch., 1995.
11. Louri A., Sung H., **A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation**, Proc. of Inter. Conf. on Parallel Processing, August 1992.
12. Mounes-Toussi F., Lilja D.J., Li Z., **An Evaluation of a Compiler Optimization for Improving the Performance of a Coherence Directory**, Proc. of Inter. Conf. on Super., July 1994.
13. O'Boyle M.F.P, Nisbet A.P., Ford R.W., **A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems**, PACT'96, October 1996.
14. O'Boyle M.F.P., Ford R.W., Nisbet A.P., **Compiler Reduction of Invalidation Traffic in Virtual Shared Memory Systems**, EuroPar'96.
15. Skeppstedt J., Stenstrom P., **Simple compiler algorithms to reduce ownership overhead in cache coherence protocols**, ASPLOS, 1999.
16. Skeppstedt J., Stenstrom P., **A Compiler Algorithm that Reduces Latency in Ownership-Based Cache Coherence**, Proc. of Parallel Arch. and Compiler Tech. 95, June 1995.
17. Skeppstedt J., Dahlgren F. and Stenstrom P., **Evaluation of Compiler-Controlled Updating to Reduce Coherence-Miss Penalties in Shared-Memory Multiprocessors**, JPDC, vol 56, 1999.

Scheduling the Computations of a Loop Nest with Respect to a Given Mapping

Alain Darte[1], Claude Diderich[2], Marc Gengler[3], and Frédéric Vivien[4]

[1] LIP, École normale supérieure de Lyon, F-69364 Lyon, France.
[2] Wannerstrasse 21, CH-8045 Zurich, Switzerland.
[3] LIM, École Supérieure d'Ingénierie de Luminy, F-13288 Marseille cedex 9, France.
[4] ICPS, Université Louis Pasteur, Strasbourg, Pôle Api, F-67400 Illkirch, France.

Abstract. When parallelizing loop nests for distributed memory parallel computers, we have to specify when the different computations are carried out (computation scheduling), where they are carried out (computation mapping), and where the data are stored (data mapping). We show that even the "best" scheduling and mapping functions can lead to a sequential execution when combined, if they are independently chosen. We characterize when combined scheduling and mapping functions actually lead to a parallel execution. We present an algorithm which computes a scheduling compatible with a given computation mapping, if such a schedule exists.

1 Introduction

When parallelizing codes for distributed memory parallel computers, it is fundamental to develop efficient strategies to distribute the workload between processors, and to distribute the data involved by these computations. Indeed, for such machines, communications between processors and global synchronizations are very expensive compared to the computation speed of the processors. The problem is to find an acceptable trade-off between the two extreme solutions, a one processor execution that involves no external communication, but sequentializes all computations, and the maximal distribution of computations that exploits all parallelism but whose performance may be damaged by too many communications or synchronizations. In the field of automatic parallelization of nested loops, this problem has been cut into two sub-problems known as the *mapping problem* and the *scheduling problem*.

The first problem is the mapping, to the different processors, of the computations (i.e., the loop iterations) and of the data elements involved by them. This mapping is usually done as follows: a first step (the *alignment* phase) defines a mapping on a δ-dimensional grid of *virtual* processors, the goal being to minimize the amount of communication overhead due to non local data references. The dimension δ is usually an input to this problem. Then, a second step (the *distribution* phase) defines a mapping of the virtual processors onto physical processors. This two-step scheme follows the same principle as in HPF. The *alignment* phase can be viewed as a way to automatically derive HPF *align*

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 405–414, 2000.
© Springer-Verlag Berlin Heidelberg 2000

directives. Different formulations of the mapping problem were studied. The mapping has been studied, in similar linear algebra frameworks, among others, by Ramanujam and Sadayappan [11], Anderson and Lam [2], Bau *et al.* [3], Dion and Robert [7], Feautrier [9], and Diderich and Gengler [6].

The second problem is the definition of a partial order for the execution of the loop iterations. This order must respect the dependences in the code. It is used to rewrite the code so as to make explicit the sequential steps (more or less the global synchronizations) required by the semantics of the code. In the context of HPF, scheduling can be viewed as a way to automatically detect *independent* directives. The main algorithms, using exact representations of data dependences, are those of Feautrier [8], and Lim and Lam [10]. Allen and Kennedy [1], Wolf and Lam [12], and Darte and Vivien [5] introduced parallelism detection and scheduling algorithms that use a conservative approximation of the data dependences.

Until now, both problems - the mapping and the scheduling problems - have generally been studied separately. In most works on scheduling, the mapping is supposed to fit well with the scheduling. However, there is no reason for a given scheduling to lead to an efficient execution, if communication costs are not taken into account. It is possible that the scheduling enforces some computations to be executed by different processors, and that this "inherent" mapping involves very expensive communications. In most works on mapping, the scheduling problem is not addressed at all: the code is supposed to be compiled, for example as an HPF program, following the owner computes rule (the processor that performs an assignment is the processor that owns the memory cell that is assigned). In the least favorable case, this may lead to very poor performances, since the order in which computations are carried out is not optimized. There is indeed no reason for a particular alignment to lead to a parallel execution of the computations, if the scheduling problem is not taken into account. It is very possible that the mapping enforces a sequential utilization of the processors when respecting the data dependences in the code.

This paper is a first step in the direction of a simultaneous solution to both the mapping and the scheduling problems. We illustrate, in Section 2, why both problems cannot be solved independently in general. Then, in Section 3, we formally state the problem of compatibility between mapping and scheduling functions. In Section 4, we present our solution on an example. In Section 5, we characterize the mappings for which there exists a compatible scheduling. Finally, in Section 6, we describe an algorithm that effectively builds a compatible scheduling for a given mapping, when one exists. We conclude with some perspectives and extensions of our results.

Note: the missing proofs and explanations can be found in [4].

2 Compatibility of Mapping and Scheduling Functions

We consider here the (uniform) loop nest presented as Example 1. Suppose that we are looking for a one-dimensional alignment of this loop nest, that is, we

consider the processors to be indexed as a vector of processors. As usually, we search for an alignment which minimizes the cost of non local memory accesses. If communicated data are not kept in memory for multiple reuse the optimal 1D-alignment maps the operations `S(I,*)` and the data elements `A(I-1,*)` to processor `P(I)` which yields three local accesses `A(I-1,J-1)`, `A(I-1,J)`, and `A(I-1,J+1)`, and one remote to store `A(I,J)` (thereby breaking the owner computes rule). This alignment is not compatible with the implicit (and shortest) scheduling given by the `DOSEQ-DOALL` form: processor `P(I)` would compute all computations `S(I,*)` at time-step `I` and would thus serialize them. Nevertheless, one can find schedules compatible with the given alignment: any schedule of the form $a\mathtt{I} + b\mathtt{J}$, with $a > b > 0$, is compatible and valid, like the function which schedules `S(I,J)` at time-step $2 * \mathtt{I} + \mathtt{J}$. In terms of program transformation, this schedule is equivalent to a loop skewing and a loop interchange. In this example, the linear part of the computation mapping is given by the vector $(1, 0)$ (i.e., `(I,J)` is mapped onto `P(I)`) and the linear part of the scheduling by the vector $(2, 1)$. The compatibility can be read from the fact that $(2, 1)$ and $(1, 0)$ are linearly independent. However, if the scheduling `DOSEQ-DOALL` is imposed, we have to change the alignment. One possible solution is to map `A(I,J)` on processor `P(J)` (thus with two non local accesses, instead of one). In this example, either the scheduling or the alignment can be chosen optimal, but the optimal scheduling and the optimal alignment are incompatible. There exist of course cases where the optimal scheduling and alignment are compatible.

Example 1.

```
DOSEQ I = 1, N
 DOALL J = 1, N
S A(I,J) = A(I-1,J-1) + A(I-1,J) + A(I-1,J+1)
 ENDDO
ENDDO
```

3 Statement of the Problem

The problems of mapping and scheduling were both mainly studied in the affine framework. In this framework, the mapping and scheduling functions are (multi-dimensional) affine functions, and the virtual processors form a grid. We suppose that we want to parallelize a loop nest while exhibiting δ degrees of parallelism. The scheduling functions must be such that the δ dimensions of the mapping are actually parallel: at each time step defined by the scheduling (in steady state) some operations are executable independently; we want the mapping to project this set of computations onto a set of processors of dimension δ. A schedule and a computation mapping which satisfy this property are said *compatible*.

As illustrated by Example 1, an alignment that minimizes the communication and a scheduling that expresses the maximum achievable parallelism can lead to a completely sequential execution when used together. We thus have to consider both problems simultaneously, or at least to solve one with respect to the other. A general approach consists in computing a "good" alignment that is compatible

with at least one possible scheduling. Indeed, the constraints on the scheduling are mandatory, while the constraints on the locality of the accesses are not. If an alignment constraint cannot be met, this means that one access will be remote and will slow down the execution speed but will not affect correctness. So, we start by computing an optimal (optimal with respect to some communication cost) alignment and we check whether there is a scheduling compatible with it. If so, we keep both the alignment and the scheduling. On the contrary, we look for the next best alignment, checking whether it is compatible with some scheduling function, and so on. We thus have to characterize what we mean by *compatible*, to characterize mappings for which there is at least one compatible scheduling, and to provide an algorithm that builds such a scheduling when it exists.

3.1 Hypotheses and Notations

We assume that the alignment problem has been solved and we make no assumption on the mapping we are given. We focus on the scheduling problem of a single loop nest that we assume to be perfectly nested, of depth n, and containing s assignment instructions. For each instruction S, the *scheduling* function assigns a (multi-dimensional) execution date to each loop iteration and is written

$$\begin{aligned} E_S\colon\ & \mathcal{D} \longrightarrow \mathbb{T}_S \\ & \boldsymbol{i} \longmapsto E_S(\boldsymbol{i}) = \mathbf{E}_S\,\boldsymbol{i} + \boldsymbol{e}_S \end{aligned}$$

where $\mathbb{T}_S$ is the d_S-dimensional time space associated with S (it is a subset of $\mathbb{Z}^{d_S}$). $E_S(\boldsymbol{i})$ is the time-step when iteration $\boldsymbol{i}$ of instruction S is scheduled (time-steps are lexicographically ordered). Similarly, for each instruction S, the *mapping* function assigns a processor to each loop iteration and is written

$$\begin{aligned} C_S\colon\ & \mathcal{D} \longrightarrow \mathbb{P} \\ & \boldsymbol{i} \longmapsto C_S(\boldsymbol{i}) = \mathbf{C}_S\,\boldsymbol{i} + \boldsymbol{c}_S \end{aligned}$$

where $\mathbb{P}$ is the virtual δ-dimensional grid of processors. $C_S(\boldsymbol{i})$ is the processor on which iteration $\boldsymbol{i}$ of instruction S is executed. All matrices $\mathbf{C}_S$ and $\mathbf{E}_S$ are assumed to be of full row rank.

There exist different equivalent criteria to define compatibility. We can say that the scheduling function E_S and the mapping function C_S of an instruction S are *compatible* if and only if at any time any virtual processor is supposed to execute a limited number of iterations that does not depend on the loop bounds (that may be parameterized). Mathematically, this is equivalent to:

$$\operatorname{rank}\begin{pmatrix} \mathbf{E}_S \\ \mathbf{C}_S \end{pmatrix} = d_S + \delta. \tag{1}$$

Indeed, if this rank is not $d_S + \delta$, there is a nonzero vector $\boldsymbol{x}$ such that $\mathbf{E}_S\,\boldsymbol{x} = 0$ and $\mathbf{C}_S\,\boldsymbol{x} = 0$. Consequently, all iterations $\boldsymbol{i}' = \boldsymbol{i} + \lambda\boldsymbol{x}$ are performed at the same time $E_S(\boldsymbol{i})$ and on the same virtual processor $C_S(\boldsymbol{i})$, whatever the integer λ. This matrix constraint is well known when applying loop transformations. Here the first dimensions correspond to time, the last dimensions to space.

3.2 The Underlying Scheduler

In this paper, we solve our problem for a particular scheduling algorithm called Darte-Vivien and fully detailed in [5]. This algorithm generalizes both the Allen-Kennedy algorithm [1] and the Wolf-Lam algorithm [12]. It works on an over-approximation of dependences by polyhedra. Here, we only state the details of Darte-Vivien needed to understand the rest of this paper. This algorithm produces, for each statement S, a multidimensional affine function: $(S, \boldsymbol{i}) \mapsto (\mathbf{E}_S^1 \boldsymbol{i} + \rho_S^1, \ldots, \mathbf{E}_S^{d_S} \boldsymbol{i} + \rho_S^{d_S})$, where d_S denotes the dimension of the schedule for S. Different statements may have different schedule dimensions. Briefly speaking, Darte-Vivien computes recursively some strongly connected subgraphs denoted $G_u(S, i)$. The graphs $G_u(S, i)$ contain some nodes which correspond to statements and which are called *actual*, and some other nodes which are called *virtual*. If $i \leq d_S$, the graph $G_u(S, i)$ is defined as the strongly connected component, containing S, of the subgraph $(G_u(S, i-1))'$ of $G_u(S, i-1)$ generated by all the edges that can not be satisfied by the first $(i-1)$ dimensions of any schedule. If a statement T is in $G_u(S, i)$, then $G_u(S, i) = G_u(T, i)$ and statements S and T have the same i-th linear part $\mathbf{E}_S^i$ in their schedules. If C is a set of edges of $G_u(S, i)$, $\boldsymbol{w}(C)$ is the sum of the dependence weights along C, and $l(C)$ is the number of edges in C whose tail is an actual node and which are satisfied by the i-th dimension of the schedule. Then the linear part of the schedule of any statement S is easily characterized: the set of admissible $\mathbf{E}_S^i$ is the polyhedron $\mathsf{P}(S, i)$: $\{X \mid \forall \mathcal{C}$ cycle of $G_u(S, i), X\boldsymbol{w}(\mathcal{C}) \geq l(\mathcal{C})\}$. Conversely, any collection of vectors $\mathbf{E}_S^i \in \mathsf{P}(S, i)$ such that $\mathbf{E}_S^i = \mathbf{E}_T^i$ for each statement T in $G_u(S, i)$ is the valid linear part of a schedule. Thus, we can explicit the set of all possible schedules (up to some regularity conditions). In this set we will look for one schedule compatible with the given mapping. Finally, let $\mathsf{VS}(S, i)$ be the vector space generated by $\mathsf{P}(S, i)$ ($\mathsf{VS}(S, i) \subsetneq \mathsf{VS}(S, i+1)$).

4 Example

In this section, we illustrate on an example how to build a schedule compatible with a given computation mapping. Here we chose a simple example for clarity. It illustrates the main lines of our technique but does not exhibit all the complexity of the problem, which appears only for some loop nests of dimension at least 3. The existence condition of a compatible schedule is presented in Section 5. The formal algorithms used to build such a schedule are presented in Section 6.

The Mappings. Figure 1 presents Example 2 and its (uniform) dependences. Here we look for a one-dimensional schedule and a one-dimensional mapping. We assume that, possibly because of other loop nests, data `a(I,J)` and operation S_1`(I,J)` are mapped onto processor `I` (the linear part of the mapping is then vector $C_{S_1} = (1, 0)$), and data `b(I,J)`, data `c(I,J)`, and operation S_2`(I,J)` are mapped onto processor `J` (the linear part of the mapping is then vector $C_{S_2} = (0, 1)$). The linear part of the schedule must be linearly independent of the mapping directions (Condition (1)). As the mapping functions are $(0, 1)$ and $(1, 0)$, we cannot use a schedule whose linear part is parallel to one of the axes.

Example 2.

```
DO I=1,N
 DO J=1,N
  S1 a(I,J) = b(I-1,J-1)+c(J,I)
  S2 b(I,J) = a(I-1,J)+a(I,J-1)+c(I,J)
 ENDDO
ENDDO
```

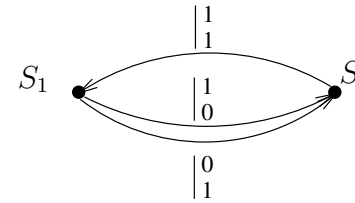


Fig. 1. Code and dependence graph for Example 2.

Constraints on the Schedule. Let vector X and constants ρ_{S_1} and ρ_{S_2} define a one-dimensional affine schedule: S_k`(I,J)` is then scheduled at time $\mathtt{X}(\mathtt{I},\mathtt{J})^t + \rho_{S_k}$, $k \in \{1,2\}$. The three dependences give three constraints on this schedule:

- S_1`(I,J)` depends on S_2`(I-1,J-1)`. Therefore, $(X(\mathtt{I},\mathtt{J})^t + \rho_{S_1})$ must be greater than $1 + (X(\mathtt{I}-1,\mathtt{J}-1)^t + \rho_{S_2})$, i.e., $X(1,1)^t + \rho_{S_1} - \rho_{S_2} \geq 1$.
- S_2`(I,J)` depends on S_1`(I-1,J)`. Therefore, $X(1,0)^t + \rho_{S_2} - \rho_{S_1} \geq 1$.
- S_2`(I,J)` depends on S_1`(I,J-1)`. Therefore, $X(0,1)^t + \rho_{S_2} - \rho_{S_1} \geq 1$.

From the previous set of constraints, we infer that the vector $X = (x, y)$ is the linear part of a valid one-dimensional schedule if and only if it belongs to the polyhedron: $\mathsf{P} = \{(x,y) \mid 2x + y \geq 2 \text{ and } x + 2y \geq 2\}$. This polyhedron generates the vector space $\mathsf{VS} = \mathbb{Q}^2$.

The Scheme. A schedule compatible with the mapping is built in three steps:

1. We build a vector $\mathbf{F} \in \mathsf{VS}$ satisfying Equation (1) for both S_1 and S_2 ($\mathsf{VS} \supset \mathsf{P}$).
2. From $\mathbf{F}$, we build a vector $\mathbf{E} \in \mathsf{P}$ satisfying Equation (1) for both S_1 and S_2.
3. We compute the constants ρ_S that, associated with $\mathbf{E}$, define a valid schedule.

Building a Solution in the Vector Space. We need a vector in the vector space $\mathsf{VS} = \mathbb{Q}^2$ which belongs neither to $\mathsf{C}(S_1) = \text{Span}\{(1,0)\}$ nor to $\mathsf{C}(S_2) = \text{Span}\{(0,1)\}$. We consider a vector in VS, but not in $\mathsf{C}(S_1)$ (resp. $\mathsf{C}(S_2)$), say $X_1 = (0,1)$ (resp. $X_2 = (1,0)$). Neither of them is a solution as $X_1 \in \mathsf{C}(S_2)$ and $X_2 \in \mathsf{C}(S_1)$. But any other vector on the line defined by X_1 and X_2 is independent with both C_{S_1} and C_{S_2}, e.g. $(X_1 + X_2)/2 = (1/2, 1/2)$. To get an integral solution, we scale this vector and we obtain: $\mathbf{F} = (1,1)$.

Building a Solution in the Polyhedron. We know a vector $\mathbf{F}$ in the vector space $\mathsf{VS} = \mathbb{Q}^2$ which is linearly independent with both the vectors C_{S_1} and C_{S_2}. What we need is a vector $\mathbf{E}$ of P with the same property. In fact $(1,1)$ belongs to P and our problem is already solved! To show how to proceed when we are not so lucky, suppose we found the vector $(1,-1)$ of VS, which also belongs neither to $\mathsf{C}(S_1)$ nor to $\mathsf{C}(S_2)$. First we consider an arbitrary vector $\mathbf{P}$ of P, e.g. $\mathbf{P} = (1,1)$ (such a vector can easily be found by linear programming [5]). We want to add λ times the vector $\mathbf{P}$ to $\mathbf{F}$ so as to obtain a vector $\mathbf{E} = \mathbf{F} + \lambda\mathbf{P}$ which belongs to P and is linearly independent with the vectors C_{S_1} and C_{S_2}. As $\mathsf{P} = \{(x,y) \mid 2x + y \geq 2 \text{ and } x + 2y \geq 2\}$, condition $(\mathbf{F} + \lambda\mathbf{P}) \in \mathsf{P}$ is equivalent to $\lambda \geq 1$. We cannot choose $\lambda = 1$ which leads to $\mathbf{E} = (2,0)$ which is collinear

with C_{S_1}. We can take $\lambda = 2$ which gives the solution $\mathbf{E} = (3, 1)$. Note that this mechanism gives in general *a* solution, not an *optimal* solution.

Computing the Constants. We have built the linear part of our schedule, say $\mathbf{E} = (1, 1)$, but we still need the constants. The constants can be computed using a shortest-path algorithm. In our example, this is not needed: the inner product of $(1, 1)$ with each distance vector is already greater than 1, so we can take $\rho_{S_1} = \rho_{S_2} = 0$. S_1`(I,J)` and S_2`(I,J)` are both computed at time `I+J`. At time `T`, processor `P` only has to execute the two operations S_1`(P,T-P)` and S_2`(T-P,P)`. Here is the code corresponding to the whole transformation:

```
DOSEQ T = 2, 2N
 DOALL P = max(T-N,1), min(N,T-1)
  S1 a(P,T-P) = b(P-1,T-P-1)+c(T-P,P)                /* on processor P */
  S2 b(T-P,P) = a(T-P-1,P)+a(T-P,P-1)+c(T-P,P)       /* on processor P */
 ENDDO
ENDDO
```

5 Existence of a Compatible Schedule

As stated in Section 3, we need to find, for each statement S and each integer i in $[1, d_S]$, a vector $\mathbf{E}_S^i$ in $\mathsf{P}(S, i)$ such that the vectors $\mathbf{E}_S^1, ..., \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, ..., \mathbf{C}_S^\delta$ are linearly independent and such that $\mathbf{E}_S^i = \mathbf{E}_T^i$ for each statement T in $G_u(S, i)$.

Lemma 1 (Existence of a Solution for Darte-Vivien**).**
Let $\mathsf{C}(S)$ denote the vector space generated by the vectors $\{\mathbf{C}_S^1, ..., \mathbf{C}_S^\delta\}$. We can associate to each statement S a sequence of vectors $\mathbf{E}_S^1, ..., \mathbf{E}_S^{d_S}$ such that:

1. $\mathbf{E}_S^i \in \mathsf{P}(S, i)$;
2. *all the statements T of $G_u(S, i)$ have the same i-th vector $\mathbf{E}_S^i$;*
3. *the vectors $\mathbf{E}_S^1, ..., \mathbf{E}_S^{d_S}, \mathbf{C}_S^1, ..., \mathbf{C}_S^\delta$ are linearly independent;*

if and only if, for each statement S, each integer i in $[1, d_S]$,

$$i + \dim(\mathsf{VS}(S, i) \cap \mathsf{C}(S)) \leq \dim(\mathsf{VS}(S, i)) \tag{2}$$

This lemma gives a necessary and sufficient condition for the existence of a schedule compatible with a given computation mapping, the underlying scheduling algorithm being Darte-Vivien. This condition states the existence of a compatible schedule iff there is one among those that Darte-Vivien can build. One could wonder whether there are examples for which there exist affine schedules compatible with the given computation mapping, but no such schedules among those Darte-Vivien can build. In fact, this cannot occur when dependence distances are approximated by polyhedra [4]. Condition (2) is a general condition.

6 The Algorithm

The algorithm, which builds the desired schedule when Condition (2) of Lemma 1 is fulfilled, proceeds in three steps: 1) building of the vectors $\mathbf{F}_S^i \in \mathsf{VS}(S,i)$ satisfying the desired properties; 2) from the vectors $\mathbf{F}_S^i$, building of the vectors $\mathbf{E}_S^i \in \mathsf{P}(S,i)$ satisfying the desired properties; 3) computing the constants ρ_S^i that, associated with the vectors $\mathbf{E}_S^i$, define a valid schedule.

6.1 Construction of the Vectors

In the algorithms listed below, each vector space is defined by one of its basis.

- Algorithm Build_Vectors takes as inputs the vector spaces $\mathsf{VS}(S,i)$ and $\mathsf{C}(S)$, and builds the desired $\mathbf{F}_S^i \in \mathsf{VS}(S,i)$ iff Condition (2) is fulfilled.

  Build_Vectors
  For $i = 1$ **to** $\max_{S \in G_u} d_S$ **do**
  For each subgraph $G_u(S,i)$ **do**
  Let $T_1, ..., T_p$ be the p statements in $G_u(S,i)$.
  $x =$In&Out$(\mathrm{Span}(\mathbf{F}_S^1, ..., \mathbf{F}_S^{i-1}) + \mathsf{C}(T_1), ..., \mathrm{Span}(\mathbf{F}_S^1, ..., \mathbf{F}_S^{i-1}) + \mathsf{C}(T_p);$
  $\mathsf{VS}(S,i))$.
  For each T in $G_u(S,i)$ let $\mathbf{F}_T^i = x$.

- Algorithm In&Out takes as input some subspaces of $\mathbb{Q}^n$, $F_1, ..., F_m$, and E. It outputs a vector $x \in (E \setminus \cup_{j=1}^m F_j)$.

  In&Out$(F_1, ..., F_m; E)$
  $x_1 =$ Find_Point_Not_In(F_1, E).
  For $i = 2$ **to** m **do**
  $y =$ Find_Point_Not_In(F_i, E).
  $H = \left\{\lambda x_{i-1} + (1-\lambda) y \mid \lambda \in \left\{0, \frac{1}{i}, ..., \frac{i}{i}\right\}\right\}$
  Choose x_i in H such that: $\forall j \in [1,i]$, Point_Is_Not_In$(x_i, F_j) =$ True.
  Return x_m.

- Algorithm Find_Point_Not_In takes two vector subspaces F and E and outputs a vector of $E \setminus F$, e.g. by testing all the vectors of a basis of E.
- Algorithm Point_Is_Not_In takes a vector x and a vector space F and outputs True if and only if $x \notin F$. This can be done by Gaussian elimination.

6.2 Construction of the Schedule Linear Parts

Preprocessing. For each statement S we complete $\{\mathbf{F}_S^1, ..., \mathbf{F}_S^{d_S}, \mathbf{C}_S^1, ..., \mathbf{C}_S^\delta\}$ in a set of n linearly independent vectors using some vectors $\mathbf{L}_S^1, ..., \mathbf{L}_S^{n-\delta-d_S}$. We build the matrix $\mathbf{M}_{S,0}$ below, where $\mathbf{F}_S$, resp. $\mathbf{L}_S$, resp. $\mathbf{C}_S$, is the matrix whose i-th row vector is the vector $\mathbf{F}_S^i$, resp. $\mathbf{L}_S^i$, resp. $\mathbf{C}_S^i$. For each graph $G_u(S,i)$ we build (e.g. by rational linear programming [5]) a solution $\mathbf{P}(S,i)$ of the system:

$$\begin{cases} e = (x_e, y_e) \in (G_u(S,i))' \Rightarrow \mathbf{P}(S,i)\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \\ e = (x_e, y_e) \notin (G_u(S,i))' \Rightarrow \mathbf{P}(S,i)\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \end{cases} \tag{3}$$

$$\mathbf{M}_{S,0} = \begin{pmatrix} \mathbf{F}_S \\ \mathbf{L}_S \\ \mathbf{C}_S \end{pmatrix}$$

We want all statements included in $G_u(S,i)$ to have the same i-th linear part, and this linear part to be a point of $\mathsf{P}(S,i)$. For that, we add to the i-th row of each matrix $\mathbf{M}_{T,i}$ the same adequate number of times the vector $\mathbf{P}(S,i)$.

Algorithm to Build Linear Parts in $\mathsf{P}(S,i)$ from Linear Parts in $\mathsf{VS}(S,i)$

For $i = 1$ **to** $\max_{S \in G_u} d_S$ **do**

- **For each** subgraph $G_u(S,i)$ **do**
  1. Find an integer ν s.t. $(\mathbf{F}_S^i + \nu\, \mathbf{P}(S,i))$ belongs to $\mathsf{P}(S,i)$, i.e. s.t. there exist some constants ρ_S satisfying for each edge $e = (x_e, y_e) \in G_u(S,i)$:

$$\begin{cases} e \in G_u(S,i)' \text{ or } x_e \text{ is virtual} & \Rightarrow (\mathbf{F}_S^i + \nu \mathbf{P}(S,i))\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} \geq 0 \\ e \notin G_u(S,i)' \text{ and } x_e \text{ is actual} & \Rightarrow (\mathbf{F}_S^i + \nu \mathbf{P}(S,i))\boldsymbol{w}(e) + \rho_{y_e} - \rho_{x_e} \geq 1 \end{cases}$$

  2. **For each** T in $G_u(S,i)$, let $\mathbf{M}'_{T,i-1}$ be equal to $\mathbf{M}_{T,i-1}$ plus the vector $\mathbf{P}(S,i)$ on the i-th row. Let $\gamma_T = \det(\mathbf{M}_{T,i-1})$ and $\gamma'_T = \det(\mathbf{M}'_{T,i-1})$.
  3. Compute the set: $\Gamma = \left\{ \frac{-\gamma_T}{\gamma'_T - \gamma_T} \;\middle|\; T \in G_u(S,i), \gamma_T \neq \gamma'_T \right\}$
  4. Let λ be an integer s.t. $\lambda \geq \nu$ and $\lambda \notin \Gamma$. **For each** statement T of $G_u(S,i)$, let $\mathbf{M}_{T,i}$ be equal to $\mathbf{M}_{T,i-1}$ plus λ times the vector $\mathbf{P}(S,i)$ on the i-th row. Condition $\lambda \geq \nu$ ensures that the i-th row of $\mathbf{M}_{T,i}$ belongs to $\mathsf{P}(T,i)$, while condition $\lambda \notin \Gamma$ ensures that $\mathbf{M}_{T,i}$ is non singular.

For each statement S, the first d_S rows of the matrix $\mathbf{P}_{S,d_S}$ define the d_S linear parts $\mathbf{E}_S^1, ..., \mathbf{E}_S^{d_S}$ of its schedule.

Note: the missing proofs and explanations can be found in [4].

6.3 Computation of the Constants

We have the linear parts of our schedule but not yet the constants. To build them, we process each graph $G_u(S,i)$ as follows (see [5, Section 7.1.2]):

1. Weight any edge $e = (x_e, y_e)$ of $G_u(S,i)$ with a new weight $\boldsymbol{w}'(e) = X\boldsymbol{w}(e) - l(e)$, where $l(e) = 1$ if x_e is actual and if $e \notin (G_u(S,i))'$, and $l(e) = 0$ otherwise.
2. Add a node S_0 and a zero-weight edge from S_0 to each node of $G_u(S,i)$.
3. Use a shortest path algorithm and let the constant ρ_S^i be the opposite of the weight of the shortest path from S_0 to S.

6.4 Algorithm Complexity

Algorithm Build_Vectors has a complexity of $O(s^2 n^4 (n + s^2))$. The building of the linear parts has a complexity of $O(sn^4 + Z)$, where Z is the complexity of Darte-Vivien (see [5] for details). For the constant computations see [5].

7 Conclusion

We have presented an algorithm that produces, for a perfect loop nest, an affine scheduling compatible with a given affine mapping of its computations. When the representation of the dependences is a polyhedral approximation of distance vectors, our algorithm succeeds whenever such an affine schedule exists. The cases of success are defined by a necessary and sufficient condition which can easily be checked. In this paper, in order to simplify things (!), we limited ourselves by using an approximation of dependences by polyhedra. But with a few tricks [4], our method can actually be extended to Feautrier's algorithm [8]. This algorithm works on an exact representation of dependences. Exact dependence analysis is feasible for static control programs with affine array access functions, which is the only type of programs most mapping algorithms work with.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, Oct. 1987.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. *ACM Sigplan Notices*, 28(6):112–125, June 1993.

[3] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In K. Pingali, U.Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and compilers for parallel computing - 7th international workshop*, volume LNCS 892, pages 46–60. Springer Verlag, 1994.

[4] A. Darte, C. Diderich, M. Gengler, and F. Vivien. Scheduling the computations of a loop nest with respect to a given mapping. Technical Report 00-04, ICPS, University of Strasbourg, France, 2000.

[5] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. J. Parallel Programming*, 25(6), 1997.

[6] C. G. Diderich and M. Gengler. The alignment problem in a linear algebra framework. In *Proceedings of the Hawaï International Conference on System Sciences (HICSS-30), Software Technology Track*, pages 586–595, Wailea, HI, Jan. 1997. IEEE Computer Society Press.

[7] M. Dion and Y. Robert. Mapping affine loop nests: New results. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing and Networking, International Conference and Exhibition*, volume LCNS 919, pages 184–189. Springer-Verlag, 1995.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, 1992.

[9] P. Feautrier. Towards automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[10] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1997.

[11] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE TPDS*, 2(4):472–482, Oct. 1991.

[12] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE TPDS*, 2(4):452–471, Oct. 1991.

Volume Driven Data Distribution for NUMA-Machines

Felix Heine and Adrian Slowik

University of Paderborn, Germany,
felix.heine@syskoplan.de, aslowik@trace.de

Abstract. Highly scalable parallel computers, e.g. SCI-coupled workstation clusters, are NUMA architectures. Thus good *static locality* is essential for high performance and scalability of parallel programs on these machines. This paper describes novel techniques to optimize static locality at compilation time by application of *data transformations* and *data distributions*. The metric which guides the optimizations employs *Ehrhart polynomials* and allows to calculate the amount of static locality *precisely*. The effectiveness of our novel techniques has been confirmed by experiments conducted on the SCI-coupled workstation cluster of the PC^2 at the University of Paderborn.[1]

1 Introduction

Clusters of workstations promise outstanding computational power at an economically attractive price. However, good static locality is a must to utilize the aggregated power of connected nodes. To give an illuminating example, we observed the execution time of the SOR-loop to be 2.1s for one of two nodes that was assigned *all* data, while it was 28.8s for the other node with *no* local data. This huge imbalance in execution time illustrates the impact of remote memory accesses and motivates the need for data transformations and data distributions that arrange for good data locality.

1.1 Problem Formulation

We use a restricted version of the HPF block-cyclic distribution model [6] starting with a *parallel* loop nest that comprises affine loop bounds and affine index functions to multidimensional arrays. The loop nest is expected to possess exactly one parallel loop. We assume that arrays are sliced into blocks along one dimension, which are then assigned to processing nodes. In the best case, any such a block is solely accessed by the processing node that *owns* the block. Hence it is the duty of data transformations to expose a regular pattern of blocks which are accessed by unique nodes. The subsequent data distribution then has to determine an assignment of blocks to processing nodes which is consistent with this

[1] This work has been supported in part by the DFG Sonderforschungsbereich 376 "Massive Parallelität – Algorithmen, Entwurfsmethoden, Anwendungen", Paderborn

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 415–424, 2000.
© Springer-Verlag Berlin Heidelberg 2000

pattern. We do not use the *owner computes rule*. We preserve the assignment of computations to processors that was computed in previous compiler steps.

In summary, for each array of a regular program we automatically derive a *unimodular data transformation* which *reshapes* the array, and a *block-cyclic data distribution* which *distributes* the array elements among the processing nodes. The distribution employs some cycle length and a certain offset.

1.2 Related Work

The topics of data transformation and data distribution have attracted great interest within the last decade, such that an overwhelming amount of publications has emerged in this field. But unfortunately, it is difficult to compare the efficiency of our approach to those described in the literature, because in the literature, locality improvements are measured in runtime improvements with regard to some specific target architecture. We instead provide a general technique to derive parametric formulas that map to the number of local and remote accesses, and the locality optimization we propose also takes place on this level. Thus we express the achieved improvements in terms of formulas which do not depend on details of the target architecture. Nevertheless, we also provide runtime improvements observed on a SCI-coupled workstation cluster.

Known techniques for optimizing data distribution either use integer programming [7], or heuristics based on reuse vectors [11], [1], resp. affinity graphs [2]. These techniques do not consider the geometry of the iteration space, but only inspect index functions and the nesting structure of loop nests. To the best of our knowledge none of these techniques uses Ehrhart polynomials [3] to guide the selection of data transformations and data distributions. In this sense our novel approach is unique. In its general outline to generate a set of candidates and to identify one of these candidates using complex mathematical reasoning it resembles the approach taken in [7]. However, the latter uses integer programming and also neglects the geometry of the iteration space.

2 Geometric Framework

We use a multi-grid application from the area of fluid dynamics to illustrate our approach. The computational kernel is a variant of the SOR-loop. It is a 2-dimensional loop nest with 5 references to array `U` and 1 reference to array `F`. We focus on references to array `U` throughout this text. The parallel program version shown in Fig. 1, has been produced by the automatic parallelizer of our prototype compiler. The loop nest exposes parallelism on its innermost level and now is subject to subsequent data locality optimizations:

In the case of 2 parallel processors and a cyclic distribution of the columns of array `U`, we observe the access pattern shown in Fig. 2. It illustrates a *default-distribution* which results in 50% remote accesses, provided iteration point (i, j) is executed by processor $P_{j \bmod 2}$. This situation can not be remedied by a data distribution, because most array elements are accessed by both processors. Since

```
DO I = 2, M+N-2
  FORALL J = MAX(1, I-M+1), MIN(I-1, N-1)
    U(S, I-J, J-1) = (F(S, I-J, J) + U(S, I-J, J-1) + U(S, I-J-1, J)
                                   + U(S, I-J, J+1) + U(S, I-J+1, J)/4.0
  ENDFORALL
END DO
```

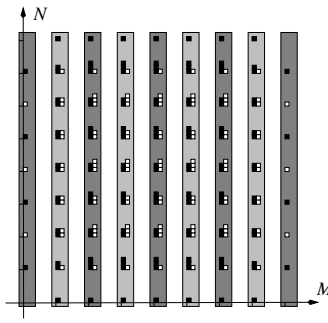
Fig. 1. SOR loop nest from multi-grid

we do not consider replication, some accesses are forced to be remote, no matter what the distribution parameters are. Nevertheless, the situation can be improved significantly by a preceding data transformation.

Now we introduce some convenient abbreviations and define fundamental geometric abstractions suited to rank transformations and distributions.

A loop nest N defines the *iteration space* $\mathcal{I}_N$. An array `X` that is accessed by a reference R_l, $l \geq 1$, defines the *index space* $\mathcal{D}_{\mathtt{X}}$. By f_l we refer to the index function of reference R_l. Furthermore, P denotes the number of processors, d the distribution dimension, B the block size, and j the parallel dimension. Henceforth, we omit subscripts if there is no risk of confusion.

It is well known that an iteration space $\mathcal{I}$ and an index space $\mathcal{D}$ both define convex polytopes [4], [8]. We represent a convex polytope $\mathcal{P}$ as usual by a set of inequations, i.e. $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{Z}^k \mid \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{C} \cdot \boldsymbol{n} + \boldsymbol{b} \geq 0\}$. The Ehrhart polynomial E of a parameterized convex polytope $\mathcal{P}$ is a function in parameters of the polytope and maps to the number of integral points contained within $\mathcal{P}$ [3]. The fundamental idea of our approach is to encode iteration points that cause local accesses by convex polytopes. Then the Ehrhart polynomials provide the means to judge the quality of a data transformation and data distribution. We employ the usual condensed notation of Ehrhart polynomials, two examples are shown in Fig. 2. The notation $E(M,N) = [10,5]_N \cdot M$ abbreviates the distinction of the two cases $E(M,N) = 10 \cdot M$ if $N \bmod 2 = 0$, and $E(M,N) = 5 \cdot M$ if $N \bmod 2 = 1$, respectively. This evaluation scheme extends to higher dimensional *cyclic-coefficients*. Moreover, a polytope may have a set of Ehrhart polynomials.



access, local memory of processor P_0

access, local memory of processor P_1

P_i	number of local accesses G_i
P_0	$\frac{1}{4}(5 \cdot M \cdot N - [10,5]_N \cdot M - [6,5]_M \cdot N) + \begin{bmatrix} 12 & 10 \\ 6 & 5 \end{bmatrix}_{N,M})$
P_1	$\frac{1}{4}(5 \cdot M \cdot N - [0,5]_N \cdot M - [4,5]_M \cdot N + \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix}_{N,M})$

Fig. 2. Accesses to array `U` and default-distribution of array `U`

In this case its polynomials are defined for convex subsets of the *parameter space*, called the *validity domains*.

Now our two primitives for data distribution, *block aggregation* and *convolution of blocks*, have to be translated into terms of convex polytopes. We begin with the primitive that addresses block aggregation. It captures the effect of data distribution with equally sized *blocks*:

Let $\boldsymbol{x}' = f_l(\boldsymbol{x}) \in \mathcal{D}$ denote the index point accessed by reference R_l at iteration point $\boldsymbol{x} \in \mathcal{I}$. Because data distribution applies to dimension d the block identified by $x'_d = \lfloor x_d/B \rfloor$ is accessed at $\boldsymbol{x}$. The non-linear expression $\lfloor x_d/B \rfloor$ can be transformed into a linear expression at the expense of an additional unknown b and a constraint C_d. If we encounter an equation containing $\lfloor x_d/B \rfloor$, we replace it by a new free variable b and additionally constrain the admissible range of x_d to satisfy C_d: $B \cdot b \leq x_d < B \cdot (b+1)$.

We proceed with the primitive for the convolution of blocks. It captures the effect of a *cyclic* data distribution. Let therefore $b = f_l(\boldsymbol{x})$ denote the expression that evaluates to the block accessed by reference R_l at iteration point $\boldsymbol{x} \in \mathcal{I}_N$. Then this block b will be assigned to processor $b \bmod P$. The expression $b \bmod P$ must be transformed into a linear expression to fit into our linear framework. We replace it by $(b - P \cdot z)$, where z is a new free variable, and additionally constrain the expression b to satisfy C_b: $P \cdot z \leq b < P \cdot (z+1)$.

Thus we can use the primitives above to describe sets of iteration points without leaving the domain of convex polytopes.

3 Data Transformation

Our method to select data transformations and data distributions can be subdivided into two phases: The first phase computes a set of *optimal* transformations and distributions for each reference *separately*. This approach is guaranteed to succeed in the case of injective index functions [5], and optimality coincides with the absence of remote accesses. The second phase ranks these candidate transformations; it compares their associated Ehrhart polynomials considering *all* references in concert and selects the *best* transformation among all candidate transformations.

Fig. 3 shows the three main steps in the generation of a candidate transformation. During step one basis vectors are selected which span subspaces of the iteration space such that these subspaces are accessed by exactly one processor (a). Then these vectors are mapped to the index space, where they span subspaces accessed by at most one processor (b). Within the next step, an unimodular transformation is determined which makes these subspaces orthogonal to one of the axes (c) [5]. Then the resulting array is sliced into blocks along the selected axis. Each of these blocks is either unused or it is used by exactly one processor, which leads to a certain *utilization pattern* of memory blocks.

Finally, an offset is determined to shift the pattern such that it matches the data distribution. The result is a transformation which turns *all* accesses performed by *one* reference into local accesses.

3.1 Ranking References

We first show how to rank a data transformation with respect to a single reference R. We start with the convex polytope of the iteration space $\mathcal{I}$ and decompose it into subspaces $\mathcal{I}_p$, such that subspace $\mathcal{I}_p$ is executed by processor P_p. Thus

$$\mathcal{I} = \{\boldsymbol{x} \in \mathbb{Z}^k \mid \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{C} \cdot \boldsymbol{n} + \boldsymbol{b} \geq 0\}$$

for appropriate matrices $\boldsymbol{A}, \boldsymbol{C}$, and a vector $\boldsymbol{b}$. The Ehrhart polynomial I of $\mathcal{I}$ maps to the number of iterations to be executed by all parallel processors. In case of the multi-grid-example shown in Fig. 1 we obtain the Ehrhart polynomial

$$I(M, N) = M \cdot N - M - N + 1$$

Because we assume a cyclic mapping of iteration points in the parallel dimension j onto a total of P parallel processors, it follows that

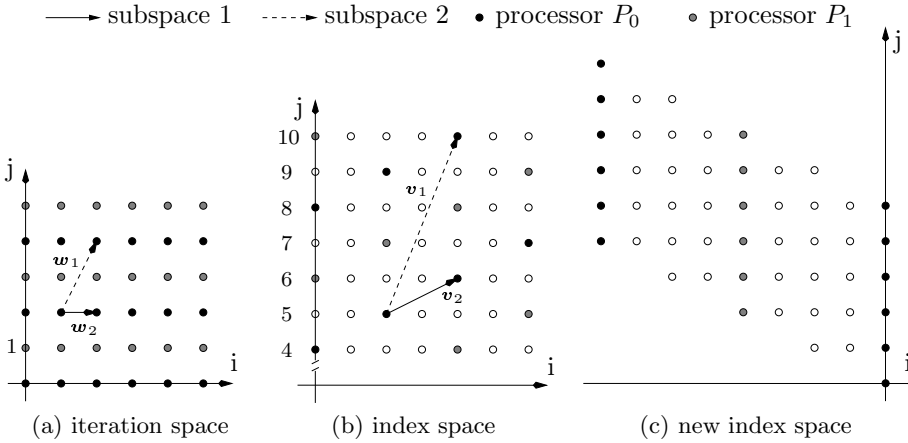
$$\mathcal{I}_p \;=\; \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{I} \;\wedge\; x_j \bmod P = p\}$$

Thus every set $\mathcal{I}_p$ also is a convex polytope, and the Ehrhart polynomial I_p of $\mathcal{I}_p$ exists. For example

$$I_p(M, N) = \frac{1}{2}(M \cdot N - M - \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}_{p,M} \cdot N + \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}_{p,M})$$

for our running example. To compute the index point within the transformed index space, we have to apply the transformation $\boldsymbol{x} \mapsto \boldsymbol{T} \cdot \boldsymbol{x} + \boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$ to the index point $f(\boldsymbol{x})$. Then we can investigate the block-cyclic distribution of the array in order to detect whether the array element $t(f(\boldsymbol{x})) = \boldsymbol{x}''$ that is accessed at iteration point $\boldsymbol{x}$ is a local array element of processor P_p. The according constraint C_p thus reads

$$C_p: \; B \cdot p \leq \pi_d(\boldsymbol{x}'') - (B \cdot P) \cdot z_2 < B \cdot (p + 1)$$



(a) iteration space (b) index space (c) new index space

Fig. 3. Steps in the generation of a candidate transformation

Note that $\pi_d(\boldsymbol{x})$ denotes the projection onto component x_d. Thus the set of iteration points $\mathcal{L}_p$ that spawn accesses to local array elements is equal to

$$\mathcal{L}_p = \{\boldsymbol{x} \in \mathcal{I}_p \mid C_p(\boldsymbol{x}) = true\}$$

We observe that the set $\mathcal{L}_p$ also is a convex polytope. The set $\mathcal{R}_p$ that spawns remote accesses is equal to the difference $\mathcal{R}_p := \mathcal{I}_p - \mathcal{L}_p$, which is not convex in the general case. Nevertheless, the polynomial of $\mathcal{R}_p$ exists and maps to the number of remote accesses. For reference R_1 = `U(I-J, J-1)` of our running example and $B = 1$, $t = id$, we obtain:

$$L_0(M,N) = \frac{1}{4}(M \cdot N - [2,1]_N \cdot M - [2,1]_M \cdot N + \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix})_{N,M}$$

Note that L_0 is a specialization of $L(M,N,p)$ with $p = 0$. Hence we can compute the polynomial $|L_0(M,N) - L_1(M,N)|$, which denotes the *imbalance* of remote memory accesses for the processors involved. In Sect. 6 we will provide further comments on the effect of such an imbalance.

3.2 Ranking Transformations

At this point we conclude that the construction of $\mathcal{L}_p$ as shown above allows to determine the local-remote access ratio of any reference R_l. We start with the iteration space as a parametric polytope, introduce a new parameter p to select iteration points executed by processor P_p and further restrict this polytope to contain only those points with local accesses. If we omit the parameter p that represents a processor P_p, we yield the desired polytope $\mathcal{L}_l$.

The volume of the polytope $\mathcal{L}_l$ is represented by an Ehrhart polynomial L_l, which serves as a metric to rank a transformation with respect to reference R_l. Thus the sum $L = \sum_l L_l$ along all references, complemented by the polynomial G that represents the total number of memory accesses, reflect the local-remote access ratio of an entire loop nest N and provides the desired metric to rank the combination of a linear data transformation and a block-cyclic data distribution.

We do not consider the case of multiple validity domains for the polytopes $\mathcal{L}_l$. In this case, one would need more information regarding the parameters of the program in order to choose the right validity domain.

3.3 Final Selection

To finally select a transformation that performs well for the entire loop nest, we *symbolically compare* the Ehrhart polynomials of different transformations and keep the best among all candidate transformations. Given a finite set of transformations, which will be constructed in Sect. 4, we compare the polynomials L of these transformations. Without further knowledge on program parameters, we first simplify periodic coefficients, i.e., we replace them by their arithmetic average, we unify program parameters, and then we compare the coefficients

$$E_M(M,N,p) = \frac{1}{4} \cdot (5 \cdot M \cdot N - \begin{bmatrix} 10 & 5 \\ 0 & 5 \end{bmatrix}_{p,N} \cdot M - \begin{bmatrix} 6 & 5 \\ 4 & 5 \end{bmatrix}_{p,M} \cdot N + \left[\begin{bmatrix} 12 & 10 \\ 6 & 5 \end{bmatrix}_{N,M}, \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix}_{N,M} \right]_p)$$

$$E_N(M,N,p) = \frac{1}{4} \cdot (6 \cdot M \cdot N - \begin{bmatrix} 12 & 6 \\ 0 & 6 \end{bmatrix}_{p,N} \cdot M - 6 \cdot N + \begin{bmatrix} 12 & 6 \\ 0 & 6 \end{bmatrix}_{p,N})$$

Fig. 4. Ehrhart polynomials associated with default-distributions

of the polynomials in descending order of their degree. Fig. 4 shows Ehrhart polynomials E_M and E_N, of our running example for 2 parallel processors. The polynomial E_M represents a default-distribution along the M-axis, whereas E_N represents a distribution along the N-axis. Both polynomials map to the number of local accesses, which implies that the data distribution along the N-axis is superior to that along the M-axis, because $\frac{6}{4} \cdot M \cdot N > \frac{5}{4} \cdot M \cdot N$ for significant problem parameters M, N. Moreover, these terms do not depend on the processor coordinate p. The distribution of array `U` along the M-axis (E_M) is illustrated in Fig. 2.

4 Enumerating Transformations

Although the result of the preceding subsection allows to rank a data transformation or a given program formulation and therefore provides a precious result by itself, we are interested in enumerating *candidate transformations* that provide locality in order to pick the best one by means of metric L.

We first search for $n-1$ vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ within the n-dimensional iteration space $\mathcal{I}$ that span disjoint subspaces of dimension $n-1$. If $\mathcal{I}_{\boldsymbol{o}}$ is such a subspace identified by some origin $\boldsymbol{o}$, i.e., $\mathcal{I}_{\boldsymbol{o}} = \{\boldsymbol{x} \in \mathcal{I} \mid \boldsymbol{x} = \boldsymbol{o} + \sum_{i=1}^{n-1} k_i \cdot \boldsymbol{w}_i, k_i \in \mathbb{Q}\}$, the following implication should hold:

$$\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{I}_{\boldsymbol{o}} \;\Rightarrow\; x_p \bmod P = x'_p \bmod P$$

Thus we intend to assign a subspace $\mathcal{I}_{\boldsymbol{o}}$ to a unique processor. In terms of generating vectors $\boldsymbol{w}_i$ we require for arbitrary iteration points $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{I}$ that

$$\boldsymbol{x}' = \boldsymbol{x} + \sum_{i=1}^{n-1} (k_i \cdot \boldsymbol{w}_i) \;\Rightarrow\; x_p \bmod P = x'_p \bmod P \tag{1}$$

Theorem 4.1 gives a sufficient condition that allows for the selection of $\boldsymbol{w}_i$. Note that below p denotes the parallel dimension of the loop nest:

Theorem 4.1 Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ denote linearly independent generating vectors from $\mathbb{Z}^n$ such that $\boldsymbol{w}_i = (w_{1,i}, \ldots, w_{n,i})^t$. Then these vectors $\boldsymbol{w}_i$ satisfy constraint (1) above, if:

i) $\forall i: \ \gcd(w_{1,i}, \ldots, w_{n,i}) = 1$ and
ii) $\forall j: \ \text{there exists at most one } i\text{: } w_{j,i} \ /= 0$ and
iii) $\forall i: \ w_{p,i} \bmod P = 0$

■

The following implication applies to the index space:

Theorem 4.2 Let $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{n-1}$ denote linearly independent vectors from $\mathbb{Z}^n$ satisfying constraint (1). Let $f(\boldsymbol{x}) = \boldsymbol{F} \cdot \boldsymbol{x} + \boldsymbol{F}_n \cdot \boldsymbol{n} + \boldsymbol{f}$ denote an index function having a square and invertible access matrix $\boldsymbol{F}$. Let further $\boldsymbol{v}_i = \boldsymbol{F} \cdot \boldsymbol{w}_i$ denote images of vectors $\boldsymbol{w}_i$ under the linear part of the index function f. Then:

$$f(\boldsymbol{x}') = f(\boldsymbol{x}) + \sum_{i=1}^{n-1} k_i \cdot \boldsymbol{v}_i \ \Rightarrow \ x_p \bmod P = x'_p \bmod P$$

■

Thus vectors $\boldsymbol{v}_i$ mentioned in Theorem 4.2 span subspaces of the index space which are accessed by at most one processor. The formulation of Theorem 4.2 implies that only those index points are involved which have a counterpart in the iteration space. Fig. 3 (a) illustrates generating vectors $\boldsymbol{w}_i$, whereas Fig. 3 (b) illustrates vectors $\boldsymbol{v}_i$ of both theorems above.

It remains to compute the transformation $\boldsymbol{T}$. Let $\boldsymbol{v}'_i = \boldsymbol{T} \cdot \boldsymbol{v}_i$ denote the image of $\boldsymbol{v}_i$ under transformation $\boldsymbol{T}$. If there exists an index j such that all vectors $\boldsymbol{v}'_i$ have a 0-entry in their jth component, then it is efficient to distribute along this dimension. We compute such a transformation $\boldsymbol{T}$ by application of Gaussian elimination combining vectors $\boldsymbol{v}_i$ into a matrix of n rows and $n-1$ columns. Its rank is $n-1$, because vectors $\boldsymbol{v}_i$ are linearly independent. Now we eliminate the entries of the last row by Gaussian elimination and place that row on a desired level. The elimination algorithm emits the transformation $\boldsymbol{T}$.

5 Data Distribution

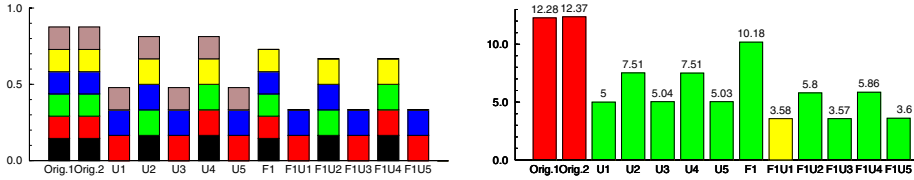
So far, we have computed the transformation matrix. It remains to determine the distribution parameters. This is done in two steps. First we determine the resulting utilization pattern, then an offset for the transformation function is computed

5.1 The Utilization Pattern

First, we introduce an important prerequisite, the notion of an array slice:

Definition 5.1 A *slice* of an array `X` with respect to a dimension d is a subspace of the index space $\mathcal{D}_{\mathtt{X}}$ which results from the evaluation of a fixed coordinate in dimension d. We say that a processor *owns* a slice $\mathcal{S}_k$, if it accesses elements within the slice but no other processor does access its elements. ■

The utilization pattern consists of slices that are owned by a specific processor and of unused slices. Using '*' to denote unused slices, we can describe the



(a) predicted performance (b) observed performance

Fig. 5. Estimated and observed performance of several multi-grid versions

pattern for the candidate transformation in Fig.3 (c) as '0,*,*,*,1,*,*,*'. We have a slice owned by processor 0, followed by three unused slices, followed by a slice owned by processor 1, etc. This pattern repeats cyclically. The blocks with unused slices always have the same size [5], in this case three.

Therefore, a simple iterative algorithm can be used to compute the pattern. Three cases must be distinguished: In the first case, the pattern fits immediately to a cyclic distribution, like the pattern '0, *, 1, *, 2, *' fits in the case of 3 processors. In the second case, a reversal transformation must be applied to the array to make the pattern fit to a distribution, which for example is true for the pattern '2, *, 1, *, 0, *'. In the third case, the pattern cannot be mapped to the distribution. Hence these transformations are removed from the set of valid candidates.

5.2 The Offset

Up to now, we just know the portion $\boldsymbol{T}$ of the transformation $t(\boldsymbol{x}) = \boldsymbol{T} \cdot \boldsymbol{x} + \boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$. The offset $\boldsymbol{T}_n \cdot \boldsymbol{n} + \boldsymbol{t}$ should be chosen such that every processor accesses those slices that it owns itself. This property is satisfied for the index function without offset. We have to determine the offset of transformation t such that it compensates the offset of f.

In the context of our simple processor-mapping the iteration point $\mathbf{0}$ is executed by processor P_0. Moreover, slice $\mathcal{S}_0$ is always owned by processor P_0. Thus it is sufficient to choose the offset such that iteration point $\mathbf{0}$ causes an access to slice S_0. Starting with $t(f(\mathbf{0})) = \mathbf{0}$ we yield $\boldsymbol{T}_n = -\boldsymbol{T} \cdot \boldsymbol{F}_n \wedge \boldsymbol{t} = -\boldsymbol{T} \cdot \boldsymbol{f}$.

6 Results and Conclusion

We have applied our techniques to a multi-grid application and investigated their impact on its execution time. Fig. 5 (a) shows ratios of remote accesses to be executed, whereas Fig. 5 (b) shows the execution time in seconds we observed on a SCI-cluster of 8 nodes and a matrix size of 512×512.

The bars from left to right represent two default versions, of which the first had all data on one node, while the second one employed a default distribution provided by the shared memory interface, 5 versions that have been optimized

for one of the 5 references to array `U`, one that has been optimized for array `F` and combinations optimized for `U` *and* `F`.

Stacked bars in sub-figure (a) are subdivided to indicate contributions of single references. Absent hatch patterns indicate that the according references do not contribute remote accesses. The real execution time observed on the SCI-cluster shown in sub-figure (b) is seen to conform strikingly well to that estimated by inspection of the remote-to-local ratio. The small deviation is due to imbalances in remote references across processors. These cause some processors to stall at synchronization barriers [5]. Compared to the worst default-parallel version, which takes 12.37s to complete, the selected version of the multi-grid application needs only 3.58s. It has been optimized for arrays `U` and `F` (bar F1U1) in concert. This gives an improvement of approx. 3.5.

From this example we conclude that our novel techniques significantly boost the performance of regular programs on NUMA-architectures. They are suited to improve data distributions of explicitly parallel programs and to guide data distribution optimizations of parallelizing compilers. Future work will address a broader set of application programs and more workstations.

Acknowledgments: We are grateful to Philippe Clauss who provided the initial implementation of Ehrhart polynomials.

References

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *PPOPP 95, Santa Clara, CA USA*, pages 166–178, June 1995.

[2] E. Ayguade, J. Garcia, M. Girones, and J. Labarta. Detecting and using affinity in an automatic data distribution tool. *Lecture Notes in Computer Science*, 892:61–75, 1995.

[3] P. Clauss. Counting Solutions to Linear and Nonlinear Constraints through Ehrhart Polynomials. In *ACM Int. Conf. on Supercomputing*. ACM, May 1996.

[4] P. Feautrier. Compiling for massively parallel architectures: a perspective. *Microprogramming and Microprocessors*, 41:425–439, 1995.

[5] F. Heine. Optimierung der Datenverteilung für SCI-gekoppelte Workstation-Cluster. Master's thesis, Universität-GH Paderborn, May 1999.

[6] C. H. Koelbel. *The High Performance Fortran handbook*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, Jan. 1994.

[7] U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, Oct. 1995.

[8] C. Lengauer. Loop parallelization in the polytope model. Technical report, Universität Passau, Fakultät für Mathematik und Informatik, 1993.

[9] A. Slowik. *Volume Driven Selection of Loop and Data Transformations for Cache-Coherent Parallel Processors*. PhD thesis, Universität-GH Paderborn, 1999. To appear (submitted).

[10] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Intitut de Recherche en Informatique et Systèmes Aléatoires, Dec. 1993.

[11] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 30–44, June 1991.

Topic 05
Parallel and Distributed Databases and Applications

Bernhard Mitschang

Local Chair

Parallel and distributed database technology is critical for many application domains. This is especially true for conventional high-performance transaction systems, but also for novel and intensive data consuming applications like data warehousing, data mining, decision support, and e-commerce. Future database systems must support flexible and adaptive approaches for data allocation, load balancing, and parallel query processing, both at the DML level and at the transaction level.

This year's Euro-Par topic "Parallel and Distributed Databases and Applications" reflects these trends by focussing on replication management and query evaluation; both topics being viewed as indispensable for modern information systems. In our session we have two papers dealing with replica mamangemnt in a direct fashion looking at algorithms and system realization issues. There is yet another paper indirectly dealing with this topic, in that this technology is among others an indispensable means to built up distributed and parallel application systems. The other remaining paper in our session focusses on issues for a novel communication infrastructure to efficiently support parallel and distributed query processing for distributed relational database management systems.

The first two papers deal with synchronous replica management. The paper by Holliday, Agrawal, and Abbadi explores the benefits of epidemic communication for replica management ensuring serializability. A detailed database simulation is used to explore the performance of the proposed protocol. The paper by Böhm, Grabs, Röhm, and Schek investigates the coordination overhead by means of an experimental assessment. Several setups that compare commercial TP-middleware-based solutions to more or less handcrafted ones are discussed. The third paper of our session by Stillger, Scheffner, and Freytag refers to another important topic for parallel and distributed database technology. The design and implementation of a communication infrastructure for an agent-based distributed query evaluation system is described. Whilst the first three papers adhere to the system and research track as mentioned in the call for papers, the fourth and last paper in our session authored by Peinl stresses the experience and application track. A case study of a large-scale online and real-time information system for foreign exchange trading is presented. Distribution, parallelism, and replica management are discussed as the underlying criteria to system efficiency assessment. In perticular it is shown how much the specific requirements of data replication and parallel processing matched with the paradigms and features

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 425–426, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of common off-the-shelf components and why proprietary solutions sometimes seemed inevitably.

All in all we can expect in the near future continued interest in research on parallel and distributed database technology and further interesting in-the-field studies on application experiences.

Database Replication Using Epidemic Communication

JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi*

University of California at Santa Barbara, Santa Barbara, CA 93106, USA
{joanne46,agrawal,amr}@cs.ucsb.edu

Abstract. There is a growing interest in asynchronous replica management protocols in which database transactions are executed locally, and their effects are incorporated asynchronously on remote database copies. In this paper we investigate an epidemic update protocol that guarantees consistency and serializability in spite of a write-anywhere capability and conduct simulation experiments to evaluate this protocol. Our results indicate that this epidemic approach is indeed a viable alternative to eager update protocols for a distributed database environment where serializability is needed.

1 Introduction

Data replication in distributed databases is an important problem that has been investigated extensively. In spite of numerous proposals, the solution to efficient access of replicated data remains elusive. Data replication has long been touted as a technique for improved performance and high reliability in distributed databases. Unfortunately, data replication has not delivered on its promise due to the complexity of maintaining consistency of replicated data. Traditional approaches for replica management incur significant performance penalties.

The traditional replica management approach requires the synchronous execution of the individual read and write operations to be executed on some set of the copies *before* transaction commit. An alternative approach is to execute operations locally without synchronization with other sites, and after termination, the updates are propagated to other copy sites [7, 8]. In this approach, changes are propagated throughout the network using an epidemic approach [8], where updates are piggy-backed on messages, thus ensuring that eventually all updates are propagated throughout the system. The epidemic approach (also called asynchronous logging) works well for single item updates or updates that commute. However, when used for multi-operation transactions, these techniques do not ensure serializability. To overcome this problem, Anderson et al. [2] and Breitbart et al. [3] impose a graph structure on the sites and classify copies into primary and secondary copies, thus restricting how and when transactions can update copies of data objects. We have developed a hybrid approach where a

* This work was partially supported by NSF grants CCR97-12108, EIA 9818320, IIS 98 17432, and IIS 99 70700

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 427–434, 2000.
© Springer-Verlag Berlin Heidelberg 2000

transaction executes its operations locally, and before committing uses epidemic communication to propagate all its updates to all replicas [1]. Once a site is sure that the updates have been incorporated at all copies, the transaction is committed. This approach ensures serializability without imposing restrictions on which sites can process update transactions or which database items can be accessed. This approach also has the advantages of epidemic propagation, namely the asynchronous propagation of update operations throughout the system which is tolerant of network delays and temporary partitions. In this paper we explore the potential benefits of epidemic communication for replica management and use a detailed database simulation to evaluate its performance.

2 System Model and Epidemic Update Protocols

We consider a distributed system consisting of a number of database server sites each maintaining a copy of all the items in the database. The sites are connected by a point-to-point network that is not required to be reliable. A transaction can originate at any site, and that site becomes the initiating or home site. Vector clocks, an extension of Lamport clocks, are used to preserve potential causal relations among operations. Vector clocks can detect if an event causally precedes, follows, or is concurrent with another event. In addition to vector clocks, each site maintains an *event* log of transaction operations. This log is not the same as the database recovery log [4] as it is used solely for epidemic communication purposes. Sites exchange their respective event logs to keep each other informed about the operations that have occurred in the system. Each site S_i keeps a two-dimensional time-table T_i, which corresponds to S_i's most recent knowledge of the vector clocks at all sites. Each time-table ensures the following *time-table property*: if $T_i[k, j] = v$ then S_i knows that S_k has received the records of all events at S_j up to time v (which is the value of S_j's local clock). When a site S_i performs an update operation it places an event record in the log recording that operation. When S_i sends a message to S_k it includes all records t such that S_i does not know if S_k has received a record of t, and it also includes its time-table T_i. When S_i receives a message from S_k it applies the updates of all received log records and updates its time-table in an atomic step to reflect the new information received from S_k. When a site receives a log record it knows that the log records of all causally preceding events either were received in previous messages, or are included in the same message.

In [1], this approach is extended to support multi-operational transactions in a database. Since strict two-phase locking [4] is widely used, we assume that concurrency control is locally enforced by the strict two phase locking protocol at all server copy sites. When a transaction, t, successfully completes its operations at the home site, S_i, it *pre-commits*. If the transaction is read-only, it can be committed at that time. Otherwise, a pre-commit record containing the readset ($RS(t)$), writeset ($WS(t)$), the values written, and a pre-commit timestamp ($TS(t)$) from the home site's vector clock is written to the local event log and the read-locks held by the transaction are released. The pre-commit timestamp is the

i^{th} row of the S_i's time-table, i.e., $T_i[i, *]$, with the ith component incremented by one. This timestamp assignment ensures that t dominates all those transactions that have already pre-committed on S_i regardless of where they were initiated. At this point there is still the possibility that the transaction will be aborted due to conflicts with other pre-committed transactions.

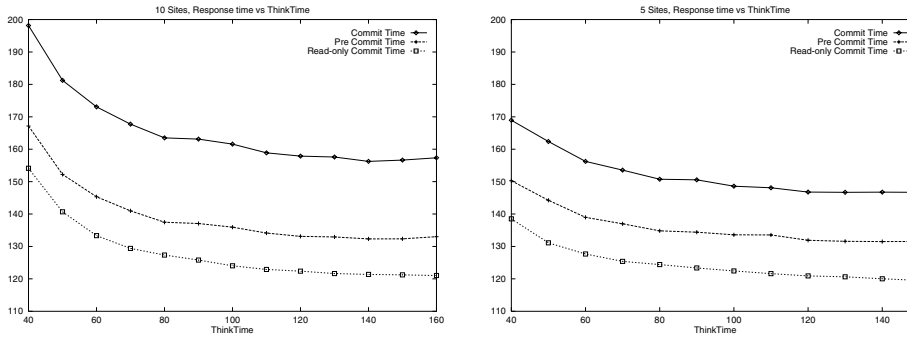
When a site S_i contacts site S_k to initiate an epidemic transfer, S_i determines which of its log records have not been received by S_k and sends those records along with S_i's time table T_i. When S_k receives the message, it reads the transaction records in order and determines if there is any conflict with transactions already in S_k's log that have not yet committed and updates its time table with the information from S_i. Two operations conflict if they are concurrent, they operate on the same data item, they originate from different transactions and at least one of them is a write operation. The vector timestamps given to pre-committed transactions can be used to determine concurrency and the read and write sets in the log records determine conflicts. If there are any conflicts, to enforce serializability both transactions involved in the conflict are aborted by releasing any locks they hold and marking the pre-commit record in the log as aborted. An aborted transaction is retained in the log and sent to other sites until it is known (via the time table) that all sites have knowledge of that transaction's termination. If the transaction t whose record was sent from S_i to S_k is not aborted, it is executed at S_k by obtaining write locks and incorporating the updates to the local copy of the database. If there are local transactions that have not yet pre-committed that hold conflicting locks, they are aborted and t is granted the locks. A transaction is committed and the remainder of its locks released when it is not aborted and it is known (via the time table) that all sites have knowledge of that transaction. This protocol ensures serializability and is explained more completely in [1]. The protocol also tolerates temporary site failures, since the information is stored in the log, and remains there until it has been received by all sites.

3 Performance Results

Our simulation [6] is based on standard database simulation models. The system and transaction parameters of the model are given in Table 1 along with their values. The generation of new transactions is governed by a parameter called

Parameter	Value	*Parameter*	Value
Number of data disks per site	1	ER – Epidemic rate	varies
CPU time needed to access disk	0.4 ms	Data disk access time	4–14 ms
Time for forced write of log	10 ms	Cache hit rate	0.8
Number of log records per page	100	Transaction read set size	5–11
Time between operation requests	10 ms	Transaction write set size	1–4

Table 1. System and Transaction Parameters



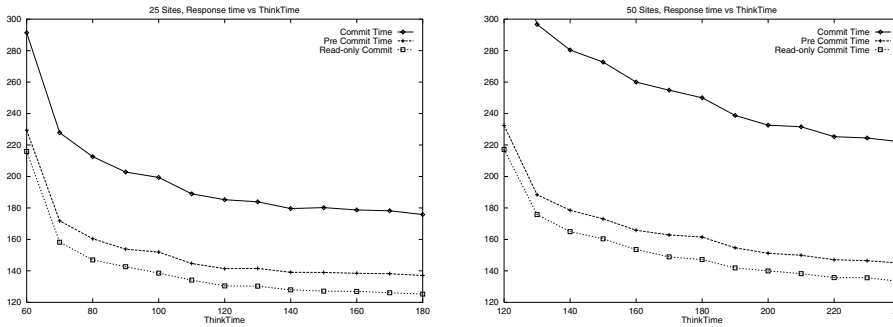
(a) Response Time for 10 Sites (b) Response Time for 5 Sites

Fig. 1. Response times as a Function of ThinkTime

"ThinkTime". This is the per site transaction interarrival time and corresponds to the "open" queuing model. The percentage of read-only transactions is 75% unless otherwise stated. When a site is ready to initiate an epidemic session with another site, it chooses a receiver site at random. The epidemic rate determines how often a site may initiate an epidemic update. All measurements in these experiments were made by running the simulation until a 95% confidence interval of 1% was achieved for each data point. All measurements of time are given in milliseconds unless stated otherwise.

3.1 Response Time Analysis

In our first set of experiments, we analyze the response time for both read-only and update transactions as a function of ThinkTime. We consider a system with 10 sites (i.e., 10 copies) and an epidemic rate of 2.0 ms. The results in Figure 1(a) show the average commit time for read-only transactions as well as both the average pre-commit and commit time for update transactions. Both the x- and y-axis are in milliseconds. Since read-only transactions execute locally, they are not adversely affected by the change in work load except at very low ThinkTime when the rate of concurrently executing transactions at each site is so high that conflicts are frequent and there is competition for resources. In fact, it is quite easy to account for the response time of read-only transactions. Each transaction has an average of 9 operations requested 10 milliseconds apart, so the issuing of the operations takes over 90 ms on average. In addition, 1.0 ms of CPU time is consumed for processing each operation, adding 9 ms to the transaction time. Disk I/O for reads takes up 16.9 ms (9 operations, 80% hit rate, 9.4 ms disk access time) for a total of 115.9 ms. At a low load, e.g., ThinkTime = 160 ms, a read-only transaction commits in about 121.0 ms. Hence a total of 5.1 ms is spent on various database management functions such as log writes, lock table



(a) 25 sites, ER = 2.0

(b) 50 sites, ER = 3.0

Fig. 2. Response time for 25 and 50 sites

management and deadlock detection, which take place during the lifetime of the transaction as well as competition with other transactions for resources.

Update transactions take longer than read-only transactions to pre-commit since they must force write update data to the recovery log disk, requiring approximately 10 ms, and the average cost of a write is slightly higher than the average cost of a read operation. However, as with read-only transactions, an update transaction pre-commits based on local execution and requires no communication. Therefore the response time for the pre-commit of update transactions closely follows the response time of read-only transactions. Committing update transactions requires communication with all the other sites in the system since the site must know that all sites have pre-committed that transaction. On average this delay which consists of disk I/O (a site which receives a pre-commit record must do the writes before putting it in its log and sending it on) and on communication costs is approximately 24 ms.

3.2 Varying Degree of Replication

Next, we compared systems with 5 (Figure 1(b)) 10 (Figure 1(a)), 25 (Figure 2(a)) and 50 (Figure 2(b)) copies. We were interested in the communications overhead introduced by the additional sites as opposed to the advantage of being able to handle more read-only transactions. Recall that 75% of the transactions generated are read-only and can thus be executed and committed at the home site. These graphs show the effect of increasing the number of sites. A ThinkTime of 100 for 50 sites means 50 transactions are started every 100 ms. Thus, to evaluate a system load of 200 newly generated transactions per second, we need to consider a ThinkTime of 50 ms for a 10 site system, a ThinkTime of 120 ms for a 25 site system and a ThinkTime of 240 for a 50 site system. In a 10 site system with 200 new transactions per second, the pre-commit time is 152.2 and the commit time is 181.2. Thus, the overhead introduced by the network to

enable the transactions to commit and ensure serializability is 19.1%. In a 25 site system the overhead is 31.2% and in a 50 site system the overhead is 71%. If we look at pre-commit times of less than 145 ms, a reasonable response time, a 5 site system can handle 100 transactions per second, a 10 site system can handle 166 transactions per second, while a 25 site system can handle 227 transactions per second and a 50 site system can successfully handle 200. After a certain point, the possibility of being able to handle more transactions by adding sites to the system is outweighed by the additional overhead introduced by those sites.

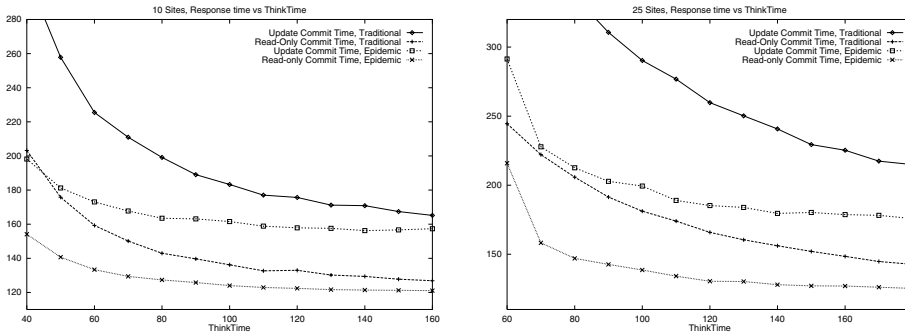
Since a lot of the time was consumed with disk I/O, we also performed experiments with a cache hit rate of 1.0, thus all read and write accesses are to the memory. Other experiments varied the transaction mix, network configuration and epidemic rate. These results are reported in [6].

3.3 Comparison with Traditional Methods

In this section we explore the advantages of epidemic based updates versus a more traditional synchronous approach. A simple traditional update protocol allows for local execution of read-only transactions just like the epidemic protocol. When an update transaction does a write, the home site DBMS must acquire write locks for that data page at each replica site. The home site DBMS sends a message to each other site requesting a write lock. When the remote site is able to grant the lock, it responds with an acknowledgment. When the home site receives acknowledgments from all other sites, it lets the transaction perform the data write and proceed with its next operation. When the transaction has completed all its operations, the home site DBMS starts a two phase commit protocol [4]. In order to assess the performance of the epidemic protocol we modeled the traditional update protocol with our simulator [6].

Experiments were performed using the traditional protocol with the same system and transaction parameters as the epidemic experiments. Response times for epidemic and traditional protocols are contrasted for 10 (Figure 3(a)), and 25 sites (Figure 3(b)). In each case, the response times are greater for the traditional than for the epidemic based approach for both read-only and update transactions. For example, in a 25 site system with a think time of 100ms, the commit response time for update transactions for the epidemic based protocol is 31% less than for the traditional approach.

The difference in read-only response time increases with increasing system load. We investigated the make-up of the read-only response times for traditional and epidemic protocols for the 10 site system (Figure 3(a)). Since the epidemic based protocol executes all write operations at remote sites together, the conflict potential and hence the blocking time between transactions is greatly reduced. We validated this hypothesis by measuring the wait time for the disk and CPU and the blocking time (the time a transaction is waiting for a lock on a data item). For example, at a ThinkTime of 120 ms, read-only transactions in the epidemic protocol spent an average of 4.2 ms waiting (for disk and CPU) and 3.9 ms blocked. The traditional protocol transaction spent 3.3 ms waiting and 12.7 ms blocked. At a higher system load, epidemic read-only transactions spent



(a) Ten sites (b) 25 sites

Fig. 3. Response time for 10 and 25 sites

8.9 ms waiting and 10.0 ms blocked while the traditional read-only transactions spent 9.1 ms waiting and 34.6 ms blocked. It was clear that the traditional read-only transactions spend more time blocked and this increases with increasing system load. This is further explained in [6].

Update transactions have a longer commit time in the traditional protocol, even at low system load. This was surprising, since, at low system load, the effects of data and resource contention would be minimal and we expected that the efficiency of two phase commit would be an advantage over the somewhat random epidemic commit process (information propagation depends on the random communication patterns among sites). We ran an additional experiment with no disk (hit rate = 1 and log disk time = 0. The results (in [6]) show that removing the effects of disk I/O has a definite effect on commit time. The commit time for an update transaction in the traditional protocol reflects two forced writes of the recovery log disk: the home site force writes its log disk before initiating two phase commit and each remote site must force its log before responding in the affirmative. The commit time for update transactions in the epidemic protocol reflects only the forced write of the recovery log by the home site; the remote sites respond after an unforced write of the pre-commit record enabling the home site to commit the transaction. A remote site forces its log later when it commits the transaction. Removing the effects of disk I/O removes the advantage of the epidemic approach in terms of commit response time at low system loads.

We also investigated the performance of the epidemic protocol in terms of total throughput. That is, how many transactions are actually committed per second. After all, good response time is meaningless if most of the submitted transactions are aborted. When the proportion of read-only transactions is 75% on a 10 site system, the throughput results [6] for the epidemic and traditional protocols are very close, although for high system load the traditional is slightly better. When only 50% of the submitted transactions are read-only, the throughput results change to favor the epidemic protocol at high system load.

4 Discussion

The epidemic protocol [1] relieves some of the limitations of the traditional approach by eliminating global deadlocks and reducing delays caused by blocking. In addition, the epidemic communication technique is more flexible than the reliable, synchronous communication required by the traditional approach. In order for an update transaction to commit in the traditional protocol, all sites must be simultaneously available and participating in the two-phase commit. In the epidemic protocol, all sites must eventually be available but need not be available at the same time. This is a great advantage in distributed systems that may experience transient failures and network congestion. This protocol has also been extended to use quorums to resolve commit decisions [5].

Current protocols include restricted execution models to ensure mutual consistency of database copies under lazy replication and protocols which allow inconsistency and non-serializable executions. We believe these limitations restrict replication to very limited classes of applications. The epidemic protocol we study in this paper [1] ensures transactional serializability and replica consistency without restricting updates or requiring reliable communication and while tolerating transient network failures. The results of our performance evaluation indicate that for moderate levels of replication, epidemic replication is an acceptable solution to guarantee serializability.

References

[1] Agrawal, D., El Abbadi, A., Steinke, R.: Epidemic Algorithms in Replicated Databases. Proceedings, ACM Symposium on Principles of Database Systems, May 1997, 161–172

[2] Anderson, T., Breitbart, Y., Korth, H.F., Wool, A.: Replication, consistency and practicality: Are these mutually exclusive? Proceedings, ACM SIGMOD, June 1998, 484–495

[3] Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S.: Update Propagation Protocols for Replicated Databases. Proceedings, ACM SIGMOD, June 1999.

[4] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993

[5] Holliday, J., Steinke, R., Agrawal, D., El Abbadi, A.: Epidemic Quorums for Managing Replicated Data. Proceedings, 19th IEEE IPCCC, Feb. 2000

[6] Holliday, J., Agrawal, D., El Abbadi, A.: Database Replication Using Epidemic Update. Technical Report TRCS00-01, Computer Science Dept. University of California at Santa Barbara, January 2000

[7] Liskov, B., Ladin, R.: Highly Available Services in Distributed Systems. Proceedings, 5th ACM Symposium, Principles of Distributed Computing, August 1986, 29–39

[8] Petersen, K., Spreitzer, M., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible Update Propagation for Weakly Consistent Replication. Proceedings, 16th ACM Symposium on Operating Systems Principles, 1997, 288–301

Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases

Klemens Böhm, Torsten Grabs, Uwe Röhm, and Hans-Jörg Schek

Database Research Group, Institute of Information Systems
ETH Zentrum, 8092 Zurich, Switzerland
{boehm|grabs|roehm|schek}@inf.ethz.ch

Abstract. We investigate the design of a coordinator for a cluster of databases. We consider the following alternatives: *TP-Heavy* using the TUXEDO TP-monitor, *TP-Lite* with the ORACLE8 database system, and a *TP-Less* coordinator implemented in Embedded SQL/C++. In particular, we investigate the scalability of full replication. We assume that update actions on all replica are executed either synchronously or asynchronously. It turns out that the *TP-Less* approach outperforms commercial TP-middleware for small cluster sizes already. Another observation is that asynchronous updates are the preferred option compared to synchronous updates. The conclusion is that a transaction protocol at the second layer must be replication-aware.

1 Introduction

The objective of the *PowerDB* project at ETH Zurich is to build a high-performance parallel database system using off-the-shelf components, notably conventional PCs and database management systems (DBMSs) and middleware that are commercially available. The project investigates how scheduling and routing on a middleware layer over a number of transactional components can be performed. The cluster components are relational DBMSs. In this current context, we also assume full replication. Replication is of advantage when the number of read operations is high, and the update rate or, to be more precise, the conflict rate, is low. At a coarse level of analysis, we can distinguish two architectures, the *symmetric architecture* and the *coordination-based architecture*. In the first case, clients are allowed to communicate with any node of the system. Gray et al. have investigated this alternative [8] and conclude that such a system may easily break down with conventional locking mechanisms. With the coordination-based architecture (cf. Figure 1), there is one distinguished node, the *coordinator*. Clients communicate only with the coordinator. It does the routing [7, 11, 13] and the scheduling. I.e., our coordinator is a second-layer transaction manager that ensures atomicity and isolation at the global level using its own locking and logging mechanisms (see [8] and references there). - This means that there is no coordinated atomic commit over all components in the style of two-phase commit (2PC) [9]. Hence, we leave aside conventional protocols for distributed transactions.

An obvious, but fundamental question now is as follows: is a larger number of components in a coordination-based architecture with full replication always better with

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 435–444, 2000.
© Springer-Verlag Berlin Heidelberg 2000

regard to throughput? At a naive level of analysis, it seems that this is indeed the case: read-only queries can go to different components. I.e., one achieves inter-query parallelism. On the other hand, updates must go to all replica. This is done in parallel. Therefore, one update action performed on all replica should not last longer than one update on a single component. The conclusion from this quick inspection is that replication improves throughput in all practical cases.

However, there is one more important consideration, and a decision must be taken: Should the parallel update actions on the replica be synchronized, i.e., should the coordinator wait until the update is performed on each component, or not, i.e. should we allow asynchronous updates? In the first case the protocol for the second layer transaction management is simpler because replication is hidden to such a protocol. In the other case, the protocol must be "replication-aware", i.e., the scheduler must know which component is already updated and which one is not, leading to a more complex multiversion protocol at the coordinator. While such a protocol is beyond the scope of this paper, it is important to investigate in quantitative terms (1) the overhead of synchronous updates, as compared to asynchronous ones, and, orthogonally, (2) the overhead of the communication infrastructure between the coordinator and its components. A series of preliminary experiments has revealed that costs of updates, when carried out synchronously, are not independent of the number of components. In fact, in this particular series of experiments, they grew almost linearly with the number of nodes! Furthermore, there are great differences with respect to the chosen infrastructure.

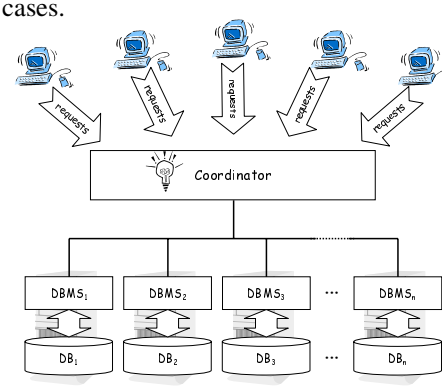


Fig. 1. Architecture of PowerDB.

This article now reports on these observations in detail and addresses the two issues from above. The contributions are as follows:

- Based on different middleware technologies – TP-Heavy, TP-Lite, and a home-grown solution, called TP-Less – we describe alternatives to run replica updates efficiently.
- We analyse the lower bounds of the coordination overhead for synchronous replication with a coordinator-based architecture.
- We compare the alternatives by means of experiments. In particular, we compare synchronous updates with asynchronous ones.

This article continues as follows: Section 2 reports on related work. In Section 3, we give an overview of the design alternatives for the coordinator. Section 4 contains their evaluation for synchronous update and compares the best results with asynchronous updates. Section 5 concludes. A longer version of this article provides more details [2].

2 Related Work

Gray et al. [8] explain that the symmetric architecture with replication does not scale well in general. Their message is that full replication with conventional locking mechanisms is not a good idea. But the analysis does not extend to a coordinator-based architecture such as the one considered here. To our knowledge, an assessment of the coordinator-based architecture in the style of [8] is not available. But it is obvious that the performance characteristics of the coordinator limit scalability.

Lazy replication alleviates the problems occurring with full replication and conventional locking mechanisms [3, 10]. Lazy replication protocols carry out updates of secondary copies in separate transactions if this does not violate serializability. Such protocols must know about the dependencies between copies. — In general, the PowerDB architecture incorporates lazy replication in a natural way. This is because serializability is not an issue with this architecture, as the coordinator takes care of it [1]. Our investigations of synchronous or asynchronous updates on replica stress the benefits of lazy replication mechanisms. Our findings also help to better understand the implications of distributed updates, as the discussion will show.

Middleware for information integration both from a functionality and a performance perspective such as [6] or [12] is a related topic. Since their motivation primarily is information integration, addressing synchronous updates is an issue of minor importance.

3 Design Alternatives

We investigate the following coordinator design alternatives: *TP-Heavy* uses a transaction monitor, *TP-Lite* deploys state-of-the-art database systems, and *TP-Less* coordinates directly via basic operating system routines.

3.1 TP-Heavy: Transaction Monitor TUXEDO

The term *TP-Heavy* denotes an approach to build an application of (distributed) components. Such an approach deploys the functionality of a Transaction Processing monitor (TP monitor) [9]. The following TP monitor features are relevant for our investigation:

Service Abstraction A TP monitor application may consist of different services. The TP monitor provides the functionality to define, to call and to execute such services [14]. Additionally, it offers location transparency.

Data Transmission Primitives TP monitors offers standard data structures and corresponding management routines to provide service invocations with input data.

For our investigations, we have built a TP monitor based synchronous replication service. It comprises two service implementations `REPLICATE` and `EXECSQL`. The purpose of `REPLICATE` is to coordinate the update of the databases in the cluster. `EXECSQL` in turn processes one such update at a specific database. In order to process an update in parallel on all databases of the cluster, `REPLICATE` asynchronously calls the

EXECSQL services. Then the EXECSQL services operate independently and in parallel at their database instance. After submitting all the calls, REPLICATE simply waits until all EXECSQL processes have completed the update.

In more detail, for each component database, REPLICATE fills a communication buffer with the SQL statement and a component identifier. The TP monitor then routes the buffer to the corresponding EXECSQL service. Each EXECSQL service holds a static connection to one of the component database systems. EXECSQL retrieves the statement, executes the statement at the database system and commits. We use Embedded SQL/C to implement this.

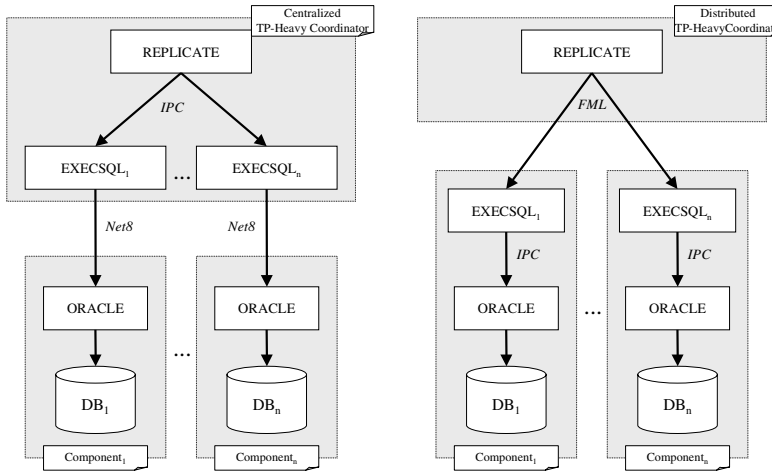


Fig. 2. Overview of TPHeavy_Centr coordinator (left), and TPHeavy_Distr (right).

An important architectural decision is where to run the EXECSQL service processes. We see the following alternatives: with *TPHeavy_Centr*, EXECSQL runs at the coordinator. As shown in Figure 2 (left), there is an EXECSQL service process for each component at the coordinator. For short, we denote this setup as TPHeavy_Centr. Note that it is important to have a separate EXECSQL process or thread for each component. Otherwise one would severely restrict parallelism. The implications are twofold:

- The communication between the REPLICATE and the EXECSQL services can use an efficient Inter-Process-Communication (IPC) protocol of the TP monitor. This is because the processes are all local to the same machine.
- The communication between the EXECSQL services and the component database systems has to go across the network. Hence, the proprietary database Client-Server communication protocol is used (ORACLE Net8 in our case).

An alternative is to run the EXECSQL service process locally at each component. Figure 2 (right) shows this configuration. We denote this with the shorthand TPHeavy_Distr. This approach has the following characteristics:

- The communication between the REPLICATE and the EXECSQL services goes across the network. This means that the TP monitor now provides the communica-

tion infrastructure for remote service calls and data transmission in fielded buffers (FML). With TUXEDO, the `BRIDGE` daemon process handles this communication.
– The communication between the `EXECSQL` services and the component database systems is local and can use efficient IPC protocols.

3.2 TP-Lite: ORACLE8 RDBMS

TP-Lite deploys the TP-monitor functionality integrated into a database system. The following features of a DBMS with TP-Lite functionality are relevant for our analysis:

Distributed Query Processing. With ORACLE8, so-called *database links* [4] give access to the relations of another ORACLE instance. These can be used like local relations when formulating SQL queries. ORACLE uses query shipping for processing such a distributed query.

4th Generation Programming Language. With PL/SQL, ORACLE provides a computationally complete database programming language that contains routines for asynchronous and parallel processing.

We have investigated ORACLE as replication coordinator employing a three-tier architecture: a dedicated ORACLE instance provides the global coordination services, which are implemented as an ORACLE PL/SQL package. This node coordinates up to n component databases. They are accessed via database links. – The replication support that is built into ORACLE is not an alternative to our approach. It is based on a deferred queue mechanism that can lead to non-serializable schedules.

The coordinator executes a given update statement on all replica. As a database link identifies exactly one remote relation, the invocation of our global coordination service results into a replication transaction, consisting of n inserts for each component:

```
replicate('INSERT INTO Orders VALUES (...)');
```

becomes

```
INSERT INTO Orders@DB1 VALUES(...)
...
INSERT INTO Orders@DBn VALUES(...)
```

However, such a single replication transaction does not give us any parallelism. ORACLE does not provide a mechanism for specifying the parallel execution of actions of the same transaction. All inserts would be executed sequentially, and the final commit would trigger the two phase commit protocol.

Thus, we have split the replication transaction into n independent subtransactions which are executed in parallel. This can be achieved using ORACLE pipes. This proprietary functionality of ORACLE8 allows for asynchronous communication between PL/SQL procedures. Since these procedures must run as independent, parallel transactions, they are

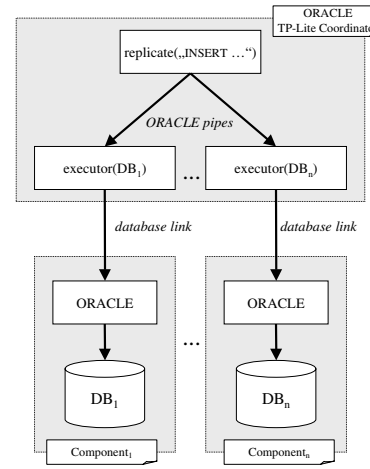


Fig. 3. `TPLite_Centr` coordinator.

executed in the context of different database sessions. Figure 3 illustrates this for an asynchronous replication service. Subsequently, we will refer to it as `TPLite_Centr`. In analogy to TP-Heavy, we have implemented the coordinator as two PL/SQL procedures, `replicate()` and `executor()`. Clients submit an update to the coordinator by invoking the `replicate()` PL/SQL procedure. It starts the execution at all components in parallel and waits for the successful end of all subtransactions. For each component, the coordinator runs a dedicated session with the `executor()` PL/SQL procedure. The executors execute their SQL command as a subtransaction using database links to the remote ORACLE instance. There is no alternative corresponding to `TPHeavy_Distr`, as we are not aware of other communication protocols for the communication between the ORACLE coordinator and its components.

3.3 TP-Less Coordinator

The third alternative is a light-weight coordinator implemented in Embedded SQL/C++ and using TCP/IP sockets for communication. In order to facilitate parallel access to the components, the coordinator is multithreaded: threads are light-weight processes, sharing the same address space. The operating system schedules threads independently. There is a dedicated thread in the coordinator for each component. The scheduler delegates the execution of SQL statements to these threads. We differentiate between two possible system architectures depending on the location of the database access, as Figure 4 shows. With the first alternative (Figure 4 (left)), called `TPLess_Centr`, the

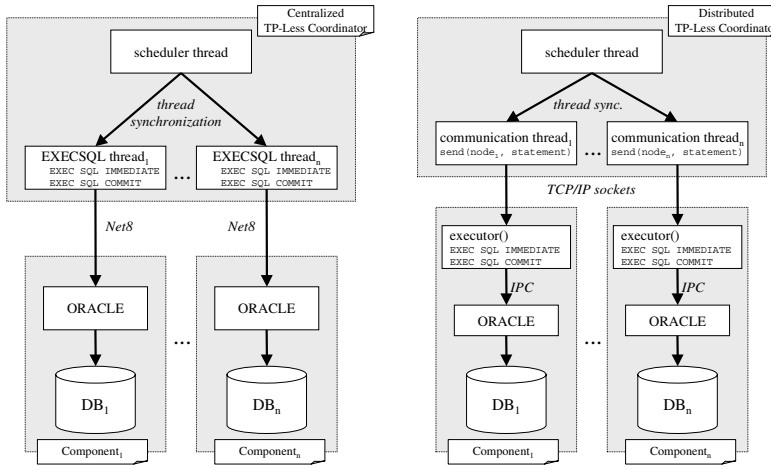


Fig. 4. Overview of `TPLess_Centr` coordinator (left), and `TPLess_Distr` (right).

threads communicate with the component database system via Embedded SQL/C. In this case, the network protocol is ORACLE's Net8 protocol. The second alternative, `TPLess_Distr` (Figure 4(right)), uses TCP/IP sockets to send the SQL statement to a corresponding executor program at the component. They locally access the database system and return the result to the coordinator.

4 Evaluation

4.1 Experimental Setup

All measurements have been carried out on a cluster of PCs (P II, 400 MHz, 128 MBytes) under Windows NT 4.0. The coordinator ran on a separate PC (P II, 400 MHz, 128 MBytes). All computers were interconnected by a switched 100 MBit Fast-Ethernet LAN. We used ORACLE 8.0.4 as component database system, and also as coordinating DBMS for the `TPLite_Centr` approach. For the approaches `TPHeavy_Centr` and `TPHeavy_Distr`, we used BEA Systems TUXEDO Version 6.5.

For all measurements, each component database was generated and populated according to the TPC-R benchmark [5] with a scaling factor of 0.1. We fully replicated the data and the indexes (notably the 3 indexes on the Orders and 7 indexes on the LineItem relation) on all nodes. The updates correspond to the TPC-R refresh functions 1, consisting of 150 inserts of new order tuples and up to 7 corresponding lineitem rows per order tuple. In total, one update stream consisted of 740 SQL INSERT statements.

4.2 Lower Bounds of Coordination Overhead for Synchronous Replication

This section reports on the lower bounds of the coordination overhead for synchronous replication with a coordinator based architecture. To conduct this analysis, we modified the distributed version of `TPLess_Distr`: the database access has been replaced by calling a wait function. However, the coordinator still "believes" to manage n components, sending SQL updates to the remote execution programs. We measured the runtime behaviour of the modified light-weight coordinator, i.e., the algorithm illustrated in Figure 5, with different numbers of nodes. The graph in Figure 5 displays the results

```
coordinator_program
for n in Nodes do in parallel
    send( start_msg, n );
    receive( reply_msg )
end for

component_program
receive( start_msg )
    Sleep( 10 ms );
reply( start_msg );
```

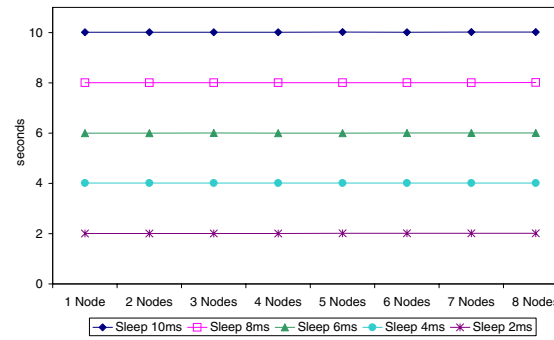


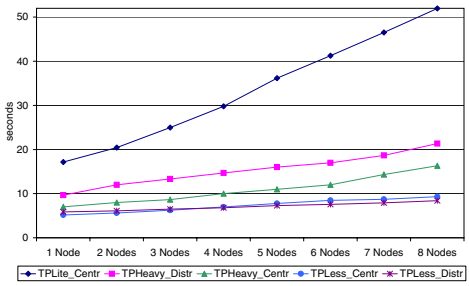
Fig. 5. Semantics and response time of the modified `TPLess_Distr` coordinator.

for different wait times. The wait function has been called 1000 times for each measurement. The result is that neither the network nor the thread synchronization of the operating system in the coordinator becomes a bottleneck, at least for cluster sizes up to 8 nodes. This is indicated by the response times with suspended execution threads (cf. Figure 5), which remain constant over the cluster size for all wait times.

4.3 Response Times of Insert Streams with Synchronous Replication

So far, our findings show that the parallel scheduling of threads executing constant-time functions does in principle scale well – even with remote invocations. We now look at scalability of synchronous replica maintenance for database systems.

The coordinator executes the updates in parallel, but synchronously in all components as discussed in Section 1. Figure 6 shows the results. All curves have a linear



	1 *node*		8 *nodes*	
	CPU load	*process size*	*CPU load*	*process size*
$\texttt{TPHeavy}_{Centr}$	27%	13 MB	75%	46 MB
$\texttt{TPHeavy}_{Distr}$	30%	7.9 MB	90%	7.9 MB
$\texttt{TPLite}_{Centr}$	65%	21 MB	95%	65 MB
$\texttt{TPLess}_{Centr}$	15%	7.0 MB	70%	7.8 MB
$\texttt{TPLess}_{Distr}$	5%	1.9 MB	30%	2.0 MB

Fig. 6. Response times and resource consumptions of coordinator approaches for TPC-R RF1.

increase in response times for an increasing number of nodes. The ORACLE coordinator `TPLite_Centr` yields the worst results. The TP-Heavy designs `TPHeavy_Distr` and `TPHeavy_Centr` perform better. The minimal coordinators `TPLess_Distr` and `TPLess_Centr` have the best response times for all cluster sizes.

`TPLite_Centr` not only proved to be the slowest solution for all node numbers. Even worse, the response time increases to 300% from 17 seconds for one node to 52 seconds with eight nodes. The reason is the very high resource consumption (cf. table of Figure 6) of this approach and the slow execution of PL/SQL procedures. These results rule out this particular TP-Lite solution.

The response times with `TPHeavy_Centr` are 30 to 40 percent better than with `TPHeavy_Distr` (and 2.5 to 3.2 times faster than `TPLite_Centr`). Recall that `TPHeavy_Centr` applies the database client-server communication protocol to communicate with the components, whereas `TPHeavy_Distr` uses TP monitor routines. Considering this, these percentages nicely show the overhead of the TUXEDO fielded buffer communication protocol compared to the ORACLE Net8 client-server communication protocol. However, both TP-Heavy approaches still show a clear increase of response time to around 230% for eight nodes compared to one node. Increasing the cluster size by one node results in a performance penalty of about 15%.

Using the minimal coordinators `TPLess_Centr` and `TPLess_Distr` does not change much: executing the update stream for eight nodes takes 180% of the response time for one node (execution at coordinator level), and 150% respectively when accessing the database directly at the components. In contrast to the `TPHeavy` approach, here it proved to be beneficial to send the SQL statements via sockets to the components and to access the database at the components directly. With cluster sizes greater than 3 nodes, the distributed version of the minimal coordinator `TPLess_Distr` is faster than the centralized `TPLess_Centr`.

The increase of response times becomes clear by looking at the resource consumption (namely average CPU load and accumulated memory allocation) of the different coordinators (cf. table of Figure 6). The values for TP-Lite and TP-Heavy tell us that the coordinator CPU is a bottleneck for 8 nodes. With `TPHeavy_Distr`, the TUXEDO BRIDGE process consumes most of the CPU time. Hence, this process and the CPU again are a bottleneck at the coordinator. While the resource consumption explains the missing scalability of Oracle- and Tuxedo-based coordinators, it is not the reason for the increased response time with `TPLess_Distr`. To see this, we must understand better how a DBMS executes single updates. This is the concern of the next subsection.

4.4 Response Times of Insert Streams with Asynchronous Replication

The primary reason for the increase of the response time is the synchronous execution of up to n updates – the coordinator must wait for all components to finish before scheduling the next update. The same update always has a slightly different duration (e.g., with our configuration, the standard deviation is about 13 ms). Hence, the coordinator tends to wait longer with each additional component. Further evaluation has shown that a probability variable with a Gaussian distribution accurately models the insertion of a tuple into a database table. We have concluded all this from one experiment with independent execution of the update streams for all nodes, i.e., asynchronous updates. If we execute the updates independently, execution time of a stream should be the average execution time per update times the number of updates. Figure 7 graphs the respective results: with `TPLess_Distr`, a stream of 740 statements behaves exactly as predicted. With `TPLess_Centr` we observe a slight increase of 15% of the one-node response time. This might be a problem of the ORACLE client library which has to synchronize the calls of the coordinator threads. In the case of `TPLess_Distr`, this problem apparently does not exist.

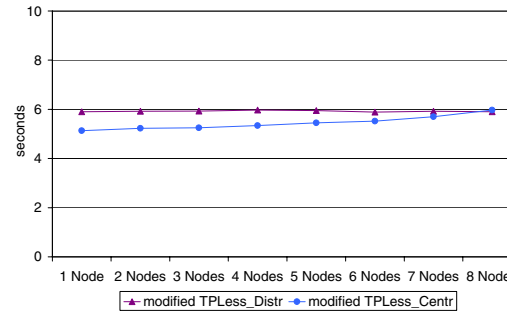


Fig. 7. Response times of independent update streams.

5 Conclusions

In the *PowerDB* project, we are developing a parallel database system using a cluster of conventional PCs and DBMSs. The PowerDB architecture is coordination-based: clients communicate with a central coordinator, which is responsible for scheduling and routing over the components. In this paper, we focus on the case of full replication. We compared three different design alternatives for the coordinator to maintain replica: *TP-Heavy* using the TUXEDO TP-monitor, *TP-Lite* via the ORACLE8 RDBMS, and a *TP-Less* coordinator using Embedded SQL/C++. In an experimental study, we investigated the scalability of these alternatives with regard to parallel, but synchronous

execution of an update stream over all replica. It turns out that the use of standard TP-middleware like TUXEDO or ORACLE may be inefficient. Even for small cluster sizes, such solutions overload the coordinator. A *TP-Less* coordinator performs better, but still yields a response time of 150% of the response time for one node. In the synchronous case, the overall execution time of an update of all replica is the execution time at the slowest component, and it grows with the number of components. This effect does not occur if we run the update streams asynchronously. This "proves" that there is almost no penalty for performing many parallel updates instead of one. With respect to coordinator overhead, our proprietary solution shows the best performance characteristics. This is because the coordinator should be as slim as possible. Commercial middleware systems with extensive functionality do not exactly have this characteristic. For our future work, we conclude that replication in large clusters requires more sophisticated decoupled replication protocols. We are currently developing such a protocol based on a multi-version concurrency control.

References

[1] G. Alonso, S. Blott, A. Feßler, and H.-J. Schek. Correctness and parallelism in composite systems. In *Proc. of the 16th Symp. on Principles of Database Systems (PODS)*, 1997.

[2] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the Coordination Overhead of Replica Maintenance in a Cluster of Databases. Technical report, Swiss Federal Institute of Technology Zurich, in preparation, 2000.

[3] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data, Philadephia, USA*, pages 97–108, 1999.

[4] Oracle Corporation. *Oracle8 Server Concepts, Release 8.0, Chapter 29*, 1997.

[5] Transaction Processing Performance Council. Tpc-r benchmark specification rev. 1.0.1. Technical report, Transaction Processing Performance Council, July 1999.

[6] F. de Ferreira Rezende and K. Hergula. The heterogeneity problem and middleware technology: Experiences with and performance of database gateways. In *Proceedings of 24th Int. Conf. on Very Large Data Bases, New York, USA*, August 1998.

[7] T. Grabs, K. Böhm, and H.-J. Schek. A document engine on a db cluster. In *Proceedings of the High Performance Transaction Systems Workshop (HPTS)*, 1999.

[8] J. Gray, P. Helland, P. E. O'Neill, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the SIGMOD Conference*, pages 173–182, 1996.

[9] J. Gray and A. Reuter. *Transaction Processing — Concepts and Techniques*. 1993.

[10] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland*, pages 126–137, 1999.

[11] U. Röhm, K. Böhm, and H.-J. Schek. Olap query routing and physical design in a database cluster. In *Proc. of the 7th Int. Conf. on Extending Database Technology (EDBT)*, 2000.

[12] U. Röhm and K. Böhm. Working together in harmony — an implementation of the corba object query service and its evaluation. In *Proc. of the 15th IEEE Int. Conf. on Data Engineering (ICDE), Sydney, Australia*, pages 238–247, March 1999.

[13] T. Tamura, M. Oguchi, and M. Kitsuregawa. High performance parallel query processing on a 100 node atm connected pc cluster. *IEICE Transactions on Information Systems, Vol. E83-D, No.1*, pages 54–63, 1999.

[14] U.K. X/Open Company Ltd. *Distributed Transaction Processing: The XATMI Specification*. X/Open Company Ltd., U.K., 1995.

A Communication Infrastructure for a Distributed RDBMS

Michael Stillger, Dieter Scheffner, and Johann-Christoph Freytag

Computer Science Department,
Humboldt University at Berlin, Germany
[stillger,scheffne,freytag]@dbis.informatik.hu-berlin.de

Abstract. We present the concept and implementation of a communication infrastructure for a distributed database system referring to the agent-based database query evaluation system AQuES. Within this model we use system components that build a federated multi agent system (MAS). We present those parts of the message transport layer that provide an "easy to handle", scalable architecture based on the "plug & play" building block principle. Furthermore, we present a generic dialog manager that enables each agent to communicate in multiple concurrent threads of execution. Based on this concept, AQuES agents keep track of a complex evaluation environment in a dynamic, multi-query scenario.

1 Introduction

Focusing on runtime query optimization for a parallel and distributed execution environment, the AQuES [6] system was designed as a multi agent system to efficiently answer SQL queries in a distributed environment. We assume query execution to take place in an open system that is subject to changing workloads and a varying agent communities competing with ordinary multi-user tasks for resources at each node and at any point in time. Dynamic optimzation is carried out to compensate unpredictable resource parameters in such environment. This overall complexity is characterized not only by data streams to be distributed in a flexible way, but also by the message flow to be managed. All components of AQuES communicate via the KQML Agent Communication Language for executing and dynamically optimizing queries, thus producing unpredictable communication flow and data flow. For this reason, a uniform, flexible, and efficient communication infrastructure is necessary.

In our AQuES system, components are computational entities that have properties like reactivity, autonomy, adaptability and goal oriented behavior [5], thus they are agents. A set of interacting software agents that cooperate in solving a global task using a facilitator is called a multi agent system (MAS). Agent communication is based on exchanging messages (KQML) [8]. We extend the facilitator concept of MAS towards a set of communicating federations, with each facilitator representing all associated agents to the rest of the system [1]. The facilitator mimics a single complex agent to other federations by integrating all services for building up a federation. Figure 1 shows such a multi federation architecture

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 445–450, 2000.
© Springer-Verlag Berlin Heidelberg 2000

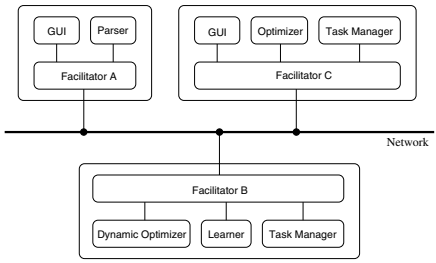
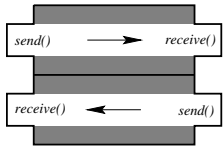


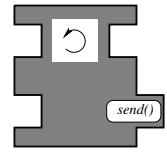
Fig. 1. Federation Architecture

with one facilitator and federation per node. Each AQuES federation might consist of one or more component agents: a graphical user interface (GUI), a parser, a static optimizer producing an optimal execution plan, a task manager, a dynamic optimizer, a learner monitoring the resources, a facilitator managing the communication flow among local components, and a communication component managing the connections to remote AQuES federations. Each federation is responsible for a local database partition and serves as a cooperative entity for the global query execution task. We now present the communication infrastructure that supports the *modular concept* of the system in a *scalable* manner for an arbitrary number of components. For more details, we refer to our technical report [7].

2 The Communication Architecture

In the AQuES message transport layer we distinguish between *intra-federation communication*, i.e., agents' communication within one federation, and *inter-federation communication*, i.e., communication among federations over the network. The following components provide the core for the AQuES communication infrastructure: Message Buffer, active Agent Adapter, Agent Plug-In, Facilitator/Router component and Network Communication Component.





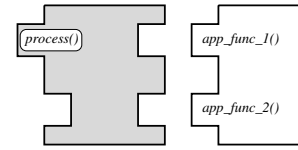


Fig. 2. Message Buffer Elem. **Fig. 3.** Agent Adapter **Fig. 4.** Agent Plug-In

The Message Buffer. This module forms the basis for intraprocess communication, i.e., it supports the exchange of messages between producers and consumers. Each agent is both a producer and a consumer of messages. For this reason, we use a Message Buffer Element (MBE) with two FIFO message queues (Fig. 2) for each instance of a component. MBEs are scalable in their sizes and each element provides two *send()*-functions and two *receive()*-functions. For processing on the same operation system, we implemented queues in shared memory, thus an MBE keeps only pointers to messages, making the transport layer a very fast medium.

The Agent Adapter. The Agent Adapter (Fig. 3) is the connecting link between the Agent Plug-In and the Message Buffer. It hides the access to a particular MBE by providing more general/global counterparts of *send()* and *receive()*. In addition, the Agent Adapter supplies a thread-based execution skeleton. Whithin any agent application can be executed with low overhead for context switches and fast concurrent message passing. The loop actively receives KQML messages and let the Agent Plug-In (Fig. 4) process (by *process()*) the messages on behalf of a particular agent.

The Agent Plug-In. The Agent Plug-In maps the application's functionality to a single single *process()* handle and is the connecting link between the application's internals and the Agent Adapter. Invoking the process method causes actions, namely application function calls and sending answer messages according to the agent's dialog protocol. Moreover, the Agent Plug-In supports the addressing of messages and the dialog management (see Sec. 3). The Agent Plug-In and the Agent Adapter give an agent its "reactive" behavior.

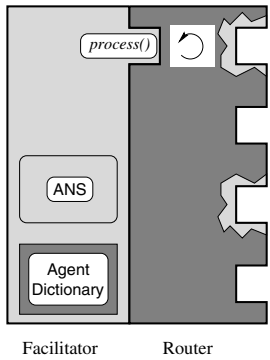


Fig. 5. Router/Facilitator

The Router. Unlike the KQML Internal Architecture [8], we only use a single router thread for the message exchange among agents on one machine, including the facilitator. The router directly receives the messages from the MBE's of all local agents. Furthermore, the router invokes the facilitator as its application for each message received, thus the facilitator runs within the router's thread (Fig. 5).

The Facilitator. The facilitator manages the dynamic (un)registration of local agents, global error handling and address resolution by means of an *Agent Dictionary*. Unlike ordinary agents, the facilitator uses *direct communication* with its MBE for sending messages [1].

Network Communication Component (NCC). The NCC extends the message transport layer by enabling inter-federation communication. The NCC is designed to appear like any other component agent in the local federation, thus providing a single message buffer interface for all remote communication links. We implemented the *Network Access* component as part of the NCC, using a freely

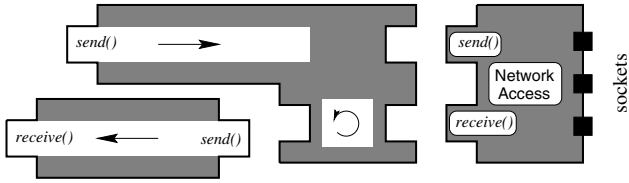


Fig. 6. Network Communication Component

available KAPI [4] software package supporting socket connectivity as an alternative to CORBA [3]. The NCC is also thread-driven. Unlike the Agent Adapter listening to the receive-port of the MBE , the NCC listens to the Network Access component for incoming messages that are buffered in a MBE queue while outgoing messages are handled through to the correct socket connection, instantly. Being a proxy to all local agents, the NCC has an agent-like appearance to the local federation provinding communication links to remote federations of the system.

3 Dialog Management

We also include the dialog management (DM) as part of the Agent Plug-In library. The DM and the concept of a *dialog* provide support for choosing the appropriate answer messages, for keeping a record which messages were sent, and for enabling asynchronious concurrent interaction with multiple partners. We explain the dialog mechanism by describing the protocol, the dialog and the dialog manager.

- Protocol: *A protocol is a state transition matrix. For each state it defines one or multiple performatives (messages identified by their performative) that the agent can accept. It defines a transition T(S, P) → (S', A) where S is the current state, P is the received performative, S' is the new state, and A is the agents action that is executed.*
  The received message is a parameter of the action function. If necessary, any reply message will be sent from inside the action function A. Upon returning from the action A the agent is in the new dialog state S' waiting for a new message. Note that the performative of the received message determines the state among multiple successor states in the protocol.
- Dialog: *A dialog D is a four tuple (I, K, P, S) where I identifies the initiating agent of this dialog. K is the dialog identifier that was created at the begin of the dialog. P is the protocol that was chosen for this dialog and S is the current state of the agent in this chosen protocol.*
- Dialog Manager: *The dialog manager of an agent consists of a set of protocols and a set of open dialogs. It provides two functions: answer and issue.*
  - *answer(msg)* is used to react to an incomming message. The dialog manager finds the appropriate dialog from the list of open dialogs identified by K. It maps the current state S of this dialog together with the received performative into a new state S' and executes the associated action A.
  - *issue(new msg)* is used by an agent to create a new dialog. The local dialog manager chooses a new protocol according to the given performative and creates a new open dialog entry and dialog id. The message is than sent out and the dialog manager of the receiving agent also opens a new dialog with the appropriate answer protocoll (see *answer*).

Table 1 shows the simplified transition matrix of a protocol for a facilitator to coordinate a query answering process. The dialog is started by a graphical user

State	*Message*	*Action*	*New State*	*Comment*
0	Evaluate	::start_evaluate()	1	//receive SQL
1	Tell	::react_parser_reply()	2	//ask optimizer
1	Sorry	::sorry_from_parser()	Finish	// SQL error
2	Tell	::send_to_taskmanager()	3	// evaluate QEP
3	Tell	::forward_stream()	3	// send result pages
3	Eos	::end_of_result()	Finish	// send last tuple and commit
3	Sorry	::sorry_from_tm()	Finish	// abort

Table 1. Dialog Protocol Matrix

interface that sends an SQL query to the facilitator. Each individual agent taking part in this protocol can start a new dialog in order to achieve its subgoal. For instance, any task manager involved might contact a dynamic optimizer agent or another task manager to resolve runtime problems (which is not shown here). For example, Table 1 shows an agent in state 3. It can accept a stream of messages from the task manager containing the result of a query (Tell), the last page of the result stream (Eos), or an error message (Sorry) indicating that the evaluation of the query plan failed. By providing a generic dialog manager in the *Agent Plug-In* block of the infrastructure, we greatly simplify the creation and integration of new component agents into the overall system. We only need to specify the protocols of an agent as well as its corresponding action functions.

4 Conclusion

The agent paradigm and MAS are appealing approaches to handle the complexity of distributed and parallel database systems including the changes of their dynamic execution environment. Within our AQuES system we extended the MAS concept towards a multi-federation concept using message passing to cope with the unpredictable data flow that can occur in dynamic query execution scenarios. We introduced an *efficient* communication infrastructure suitable to support the *modular concept* of AQuES and to provide a *scalable* architecture for any number of components. Building blocks of the message transport layer were designed to smoothly integrate intra- and inter-federation communication and to implement a generic agent execution framework. Moreover, we presented a dialog infrastructure that enables asynchronous and concurrent communication flow among agents.

References

[1] Genesereth, M. R. and Singh, N. P. and Syed, M. A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation. In *International Journal of Cooperative Information Systems*, volume 4, pages 339–367, 1995.

[2] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, volume 6(1), pages 120–135, February 1994.

[3] The Object Management Group. *CORBA/IIOP 2.2 Specification(98-7-01)*. OMG, `http://www.omg.org/`, 1998.

[4] Jay Weber, EIT. `ftp.eit.com/pub/shade/kapi*` see also: `http://hitchhiker.space.lockheed.com/aic/shade/software/KAPI`.

[5] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. In *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[6] M. Stillger, J. K. Obermaier, and J.-C. Freytag. AQuES: An Agent-based Query Evaluation System. In *Proc. Int'l. Conf. on Cooperative Information Systems*, Charleston, SC, USA, June 1997.

[7] Michael Stillger, Dieter Scheffner, and Johann-Christoph Freytag. A Communication Infrastructure for a Distributed RDBMS. Informatik Bericht 137, Computer Science Department, Humboldt University at Berlin, Berlin, Germany, 2000.

[8] Tim Finin and Richard Fritzson, Don McKay and Robin McEntire. KQML as an Agent Communication Language. In *The Proc. of 3'rd Int'l Conf. on Information and Knowledge Management (CIKM'94)*. ACM Press, November 1994.

[9] Yun Wang. DB2 Query Parallelism: Staging and Implementation. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, Zurich, Switzerland*, pages 686–691. Morgan Kaufmann, 1995.

Distribution, Replication, Parallelism, and Efficiency Issues in a Large-Scale Online/Real-Time Information System for Foreign Exchange Trading

Peter Peinl

Department of Computer Science
University of Applied Science Fulda, Marquardstraße 35, D-36039 Fulda, Germany
Institute of Parallel and Distributed High-Performance Systems (IPVR)[1]
University of Stuttgart, Breitwiesenstraße 20-22, D-70565 Stuttgart, Germany
pepe@informatik.uni-stuttgart.de

Abstract. This paper describes the design and implementation of a large-scale investment banking information system, currently used by hundreds of foreign exchange (FX) traders. It is a typical example of a distributed client/server application in a banking environment. It is shown, how much the specific requirements of data replication and parallel processing match with the paradigms and features of common off-the-shelf software components and why proprietary implementation sometimes seems inevitable and how the properties of the application, combined with performance requirements lead to the specific distribution of functionality and processing between the client and server side.

1. Introduction

This paper outlines the design and implementation of a large-scale online/real-time information system for the FX division of a major world-wide investment bank, that is actually used by hundreds of traders in a closely integrated group of trading rooms world-wide. In the design phase major issues of parallelism, the representation and location of data, their replication (timely and consistent), the distribution of work between client and server and the common issues of reliability, availability, accountability, etc. had to be considered and solutions had to be found. As the system comprises more than half a million lines of code, it is impossible to deal with all aspects in the scope of this paper. Section 2 briefly introduces the application, i.e. FX trading, and states major requirements. Section 3, outlines the system architecture and Section 4 focuses on some of the technical highlights of the system. Finally, Section 5 summarises and points to some of the lessons learnt.

[1] Work done during sabbatical at IPVR.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 451-454, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2. Application and Requirements

FX trading [1] is concerned with the exchange of the currencies of different countries. In inter-bank trading typically minimum amounts worth several million dollars are exchanged in a single trade. Apart from a multitude of existing currencies, there are different types of trade, e.g. spot, forward and option contracts [2]. The prices of different contracts are interrelated by a mathematically complex FX calculus which is implemented by the system described. The system runs in one or more trading rooms, each housing a few hundred traders. Each workplace is equipped with a powerful workstation running UNIX, and two or three large colour monitors to display (mostly real-time) information. Communication software uses the Internet stack of protocols (TCP/UDP/IP). Locally, i.e. in the trading room, a high-speed local area network (hundreds of Mbit/sec) is employed. Continuous trading is enabled by trading rooms in several continents being connected by a private wide area network.

The overall goal of the system is to provide the FX traders with all the basic rates (for standard products) in the FX market plus a powerful financial mathematics calculus. It computes the prices of non-standard FX products, for which no market prices are available and/or to incorporate some confidential pricing models. Basic rates are mostly taken from real-time market feed(s) and after some modifications relayed to the trader workstations. Those modifications reflect the trading policy and are determined by authorised traders. Once the policy has been altered, the stream of real-time data relayed to the workstations has to be altered accordingly and mirrored on the workstations with as small a delay as possible.

The decentralised organisation of several largely independent trading rooms, among others resulted in the following system requirements:

Autonomy of the trader: the entire FX calculation functionality has to be made available on every workstation; each workstation can partly or entirely disconnect from and later reconnect to the real-time rate distribution mechanism to perform calculations based on some or all trader specified rates to evaluate what-if scenarios.

Centralised policy making: trading policy is influenced by rules and parameters applied to market rates and calculation models; information has to be delivered to all workstations without loss, duplication or reordering as rapidly as possible.

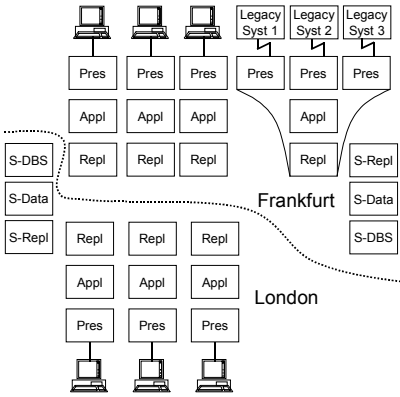
Shared up-to-date information: all the workstations in a trading room have simultaneous access to real-time market rates, which may be modified due to central policy setting; changes to the latter must not be lost during regular operation.

Recoverability: in case of system failures on a single trader workstation or the central policy setting instance, recovery should be fast, automatic and transparent to the user; in particular, policy related information must not be lost due to a failure of the system.

Different *coupling modes* between *trading rooms*: different sets of basic rates and real-time market feeds are used in each trading room, yet certain aspects of the policy are common for all trading rooms; the system has to provide the mechanisms to allow for the replication of these aspects.

3. System Architecture

The functional view of the overall architecture is sketched in the figure. It depicts two interconnected instances (trading rooms) of the FX system. The client (trader workstation) side consists of the following 3 layers. The *presentation layer* comprises all input and display functionality, but none related to the FX calculus. The *application layer* implements the FX calculus. Because of its object-oriented design, this layer effectively shields the presentation layer from all the complexities of the FX calculus. The highlight is a dynamic, graph-based, on-demand, real-time recalculation scheme. All objects are made accessible to the presentation layer by means of a publish-and-subscribe [3] interface. The primary task of the *replication layer* is to guarantee that the application layer always sees up-to-date basic rates and trading policy determining parameters, i.e. the replication layer implements a particular kind of a shared global memory.



On the server side, there are also 3 layers. The *replication layer* acts as the counterpart to the respective client layer. The *data layer* maps objects to a representation in the relational data model [4]. Mostly for organisational reasons, it was decided to use a commercial RDBMS to hold the persistent parts of the FX data. The system heavily relies on the integrity preserving functions of a DBMS to aid recovery, enable audit, etc.

Neither a single system on the market nor an easy combination of standard tools could be identified, that would technically fulfil all or enough of the given requirements. No system would support the specific coupling and replication trading rooms. Thus it was decided to build some critical mechanisms and components as a proprietary solution, but to employ off-the-shelf commercial software wherever feasible.

4. Implementation Aspects

A paramount decision was where and how to perform the calculations that reflect all the intricacies of the FX domain. In addition, the problem of dealing with more or less continuous real-time updates of the basic rates had to be solved. Centralised calculation of all possible rates was ruled out because of the expected load generated on the server, the futility of calculating values that would not be needed by any of the client instances, and the inappropriately high amount consumption of network bandwidth. Hence, by design all the calculations were situated on the client side. To further minimise the amount of work, only those values are recalculated that depend on changed input. To achieve this, structural and mathematical dependencies between the various FX products were worked out and all objects representing the financial

products were organised into an acyclical graph. As increasingly complex financial products build on each other, the height of the graph can easily surpass 10. On all workstations, the leaf nodes of this graph, i.e. the base objects, are maintained in an identical and consistent state by the replication layer. Dependent objects are dynamically recalculated on demand. The virtue of the mechanism [6] lies in its object-oriented implementation, which consists of two parts. Firstly, each objects inherits and implements abstract methods for the recalculation, connection and disconnection to the overall graph. Secondly, there is a general engine, which drives the evaluation by first arranging the objects concerned into layers and then invoking the recalculation methods. Because of this, the system can be extended easily to include new object types.

Another highlight is the local (intra-trading room) replication mechanism. Its primary task is to provide each client instance with an up-to-date replica of all the basic objects. This comprises a fast and efficient mechanism to establish an initial state on a client workstation at start-up or after recovery and the swift relay of all changes forwarded by the server instance. Commercial products examined [4] did not provide the functionality required, because among other reasons, they either lacked a broadcast/multicast feature or were difficult to adapt to our object model. Thus it was decided to implement a mechanism specific to the needs of the FX system.

5. Summary and Conclusions

System design and implementation involved complex issues of distribution, parallelism and replication. Though an attempt was made to employ as many common off-the-shelf components as possible, in some cases a critical feature was missing (at least when our development would be in full swing), which left no choice but a proprietary implementation. Often it was possible to lean heavily on well-published algorithms or their basic ideas [5]. On the other hand, relying on commercial products certainly helps in reducing the complexity of the software developed in-house, but savings sometimes turn out lower than expected at first glance.

References

1. Luca: Trading in the Global Currency Markets, Prentice Hall, 1995
2. Derosa: Options on Foreign Exchange, John Wiley & Sons, 1999
3. Chan: Transactional Publish/Subscribe: The Proactive Multicast of Database Changes, in: ACM SIGMOD Conference, 1998, p. 521
4. Freytag, Manthey, Wallace: Mapping Object-Oriented Concepts into Relational Concepts by Meta-Compilation in a Logic Programming Environment, in: AOODS, LNCS, Springer, Vol. 334, pp. 204-208
5. Birrel, Schiper, Stephenson: Lightweight Causal and Atomic Group Multicast, in: ACM Transactions on Computer Systems, Vol. 9, pp. 272-314, August, 1991
6. Peinl: Distribution, Replication, Parallelism and Efficiency Issues in a Large-Scale Online/Real-time Information System for Foreign Exchange Trading, Technical Report, IPVR, University of Stuttgart, 1999

Topic 06
Complexity Theory and Algorithms

Friedhelm Mayer auf der Heide, Mirosław Kutyłowski, and Prabhakar Ragde

Topic Chairmen

This workshop embraces algorithmic and complexity theory issues in parallel computing. A total of 10 submissions were received. Three papers were accepted, two as regular papers, one as a research note.

All papers presented during the workshop present novel algorithmic techniques for some fundamental problems. The solutions contributed by the papers settle new upper bounds for these important questions.

The first paper, "Positive Linear Programming extensions: Parallel Complexity and Applications", by Pavlos Efraimidis and Paul Spirakis, addresses the problem of approximation schemes for linear programming problem. Since the general setting of the problem is P-complete, and hence probably not suited for parallel computation, restricted versions are considered. The authors study extensions of positive linear programming problem, which still admit efficient parallel approximation and comprise many combinatorial optimization problems.

The second paper, "Parallel Shortest Path for Arbitrary Graph", by Ulrich Meyer, Peter Sanders is devoted to the problem of work-efficient parallel algorithms for the shortest path problem. The solution presented is a refinement of the Δ-stepping technique of the authors. With an algorithm for efficient search of a good step width, the authors achieve an improvement over runtime within the class of linear-work algorithms.

The third paper,"Periodic correction networks", by Marcin Kik studies the problem of sorting data that is almost sorted. However, the computation model is very restricted: only parallel compare-exchange operations are executed. Moreover, these operations form a cycle of a constant period. The problem statement is motivated by the real world sorting problems, where often the data is only slightly distorted from the sorted state, and from the efforts to design algorithms suitable for cheap hardware implementation. Surprisingly, the author achieves in this model runtime $O(\log(n))$, provided that the number of distortions from a sorted sequence in the input is also $O(\log(n))$.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 455–455, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Positive Linear Programming Extensions: Parallel Complexity and Applications*

Pavlos S. Efraimidis** and Paul G. Spirakis

Computer Technology Institute, Dept. of Computer Engineering and Informatics, University of Patras, Riga Feraiou 61, 26221 Patras, Greece,
{efraimid,spirakis}@cti.gr
http://www.cti.gr

Abstract. In this paper, we propose a general class of linear programs that admit efficient parallel approximations and use it for efficient parallel approximations to hard combinatorial optimization problems.

1 Introduction

One of the foremost paradigms in the design and analysis of approximation algorithms to hard combinatorial optimization problems [9] is:

1. First the problem is formulated as an integer program (IP),
2. then a fractional solution is found with linear programming (LP), and
3. finally the fractional solution is rounded to an integer approximate solution.

The above methodology (M) is used in many sequential approximation algorithms and hence a parallelization of the components of (M) would lead to a large number of parallel approximation algorithms for hard combinatorial problems. However the general LP problem is likely to be an inherently sequential problem, since it is complete for the class P ([2]). Moreover, any constant factor approximation to LP is also P-complete ([10],[3]). The largest, in several aspects, general class of linear programs that admits efficient parallel approximation schemes is the class of positive linear programs (PLP).

A PLP is a linear program in packing or covering form where all coefficients of matrix A and vectors b and c are non-negative (Figure 1). Solving PLP optimally is a P-complete problem ([12]), however PLP is easier to approximate than general linear programming. Luby and Nisan presented in [6] an efficient parallel (NC) approximation scheme for PLP and later Bartal et al. [1] presented a modified version for the same algorithm for distributed settings. Using PLP in methodology (M) can lead to efficient efficient parallel approximation algorithms

* An extended version of this work is given in [7].

** Financial support from the Bodosaki Foundation to perform doctoral studies is gratefully announced. Bodosaki Foundation, Leoforos Amalias 20, 10557 Athina, Greece

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 456–460, 2000.
© Springer-Verlag Berlin Heidelberg 2000

PLP in packing form	PLP in covering form
$\min \sum_{i=1}^{m} c_j x_j$ subject to	$\max \sum_{j=1}^{n} b_i y_i$ subject to
$\forall i : \sum_{j=1}^{n} a_{ij} x_j \leq b_i$	$\forall j : \sum_{i=1}^{m} a_{ij} y_i \geq c_j$
$\forall j : x_j \geq 0$	$\forall i : y_i \geq 0$

Fig. 1. The Positive Linear Programming (PLP) model. All entries of matrix A and vectors b and c are non-negative. Note that the two forms of PLP correspond to the primal and the dual of the same problem.

to problems that admit PLP formulations[1]. However the syntax of PLP is rather weak and hence only a very limited number of combinatorial optimization problems (including matching in bipartite graphs and set covering - [6]) admit native PLP formulations. We discuss the conditions for extending PLP in a natural way so that the extended model

- can model a larger number of combinatorial optimization problems, and
- at the same time still admits efficient parallel approximations.

2 Extended PLP

An extended positive linear program (ePLP) is a PLP with a number of violations that are equality constraints, covering constraints (for ePLP is packing form) and variables with negative coefficients in matrix A. As we show in Section 5 the ePLP can model a larger number of combinatorial optimization. Considering the complexity of ePLP an almost surprising result shows that:

Theorem 1 *Extensions of PLP with even one equality or covering constraints are P-hard to be approximated within any constant factor.*

The proof is based on a reduction from the Circuit Value Problem similar to the one used in [12]. Hence efficient parallel approximations are unlikely to exist even for the very simple ePLP. However we will show an algorithmic framework, the Lagrangian Search Method, that achieves efficient parallel approximations to ePLP problems if certain problem constraints can be violated by at most an arbitrary small constant factor.

3 The Lagrangian Search Method

As its name suggests, the Lagrangian Search Method (LSM), uses the general strategy of Lagrangian relaxation, that is to relax specific problem constraints by bringing them into the objective function of the linear program. The main idea is, given an ePLP to build a corresponding PLP and then to approximate

[1] There must also be an efficient parallel rounding procedure for the fractional solution. However in most cases there exists such a rounding procedure.

the PLP with the approximation scheme of [6]. However due to the limitations of the PLP model, and the fact that PLP can only be approximately solved, applying simple Lagrangian relaxation is not appropriate. LSM transforms the ePLP to an PLP in the following way:

1. All violations all transformed to appropriate combinations of packing and equality constraints so that the only violations are equality constraints.
2. The equality constraints are relaxed to packing constraints and a corresponding term is added to the objective function.

It is easy to show that the resulting PLP is equivalent to the original ePLP problem, if both problems are solved optimally. However a simple approximation to the PLP cannot guarantee an equivalent approximation to the original ePLP, since the error in the individual terms of the objective function might differ significantly from the overall approximation ratio.

The novel approach of LSM to the above limitation is the addition of Z, a new parameter, to the PLP. The modified PLP is now called PLP(Z) and instead of solving the optimization version of ePLP, PLP(Z) aims at solving only the decision version of ePLP. The role of parameter Z is critical:

- Z is an estimation of the optimal objective value of the ePLP.
- A constraint assures that in PLP(Z) the value of the original objective function of ePLP does not exceed Z.
- Knowing upper bounds for all terms of the objective function, permits the algorithm to force equidistribution of the objective function to all its terms.

Theorem 2 *In LSM, approximating PLP(Z) corresponds to solving a relaxed decision procedure for the original ePLP problem with the condition that specific problem constraints of ePLP can be violated by at most an arbitrary small constant factor.*

4 Searching with Decision Problems

In sequential algorithms solving a relaxed decision problem is generally equivalent to approximating the corresponding search problem. However in parallel algorithms this is not always true. The problem determining the parallel complexity of a search problem assuming an efficient solution to the corresponding decision problem is called the problem of "parallel self-reducibility" ([3],[4]). In LSM, the relaxed decision problem is repeatedly solved within a binary search procedure until the largest possible value of parameter Z is found. The initial upper and lower bound of the binary search procedure must be valid upper and lower bounds on the optimal objective value of the ePLP. Clearly, if LSM is considered an approach for solving general ePLP programs this is a significant limitation of the method, since there must be polynomially related upper and lower bounds on the optimal value of ePLP, for LSM to run in polylog time.

However when LSM is used in combinatorial optimization the parallel self-reducibility can be achieved in most cases. The reason is a combinatorial property that we identify on many combinatorial optimization problems.

Definition 1 *We say that a combinatorial optimization problem has the poly-bottleneck property if its optimal objective value is always within a polynomial factor of one of its input weights.*

The poly-bottleneck property holds for all problems considered in this work and interestingly, seems to be valid for a surprisingly large number of combinatorial optimization problems. The problems considered in this work, but also the k-median, the traveling salesman and many other problems have the poly-bottleneck property. The proof is usually a simple combinatorial argument. Given a hard problem with a relaxed decision procedure and the poly-bottleneck property, a 2-level binary search procedure can find an approximate solution in at most a poly-logarithmic number of steps. At each step LSM is used to decide if the current value of parameter Z is feasible for the original ePLP.

5 Applications

The PLP extensions and the LSM algorithm have been used for efficient parallel approximations to a number of hard combinatorial optimization problems. The problems are Global Routing in Gate Arrays (GRGA [8]), Scheduling Unrelated Parallel Machines (SCHED [5]), Generalized Assignment (GAS [11]), and extensions of Simple k-Matching in Hypergraphs. The algorithms run in poly-logarithmic time on a polynomial number of processors and achieve logarithmic or poly-logarithmic approximation guarantees. These are, to our knowledge, the best known parallel approximation results for the corresponding problems.

References

1. Y. Bartal, J. Byers, and D. Raz. Global optimization using local information with applications to flow control. In *38th IEEE FOCS*, pages 303–312, 1997.
2. D. Dobkin, R.J. Lipton, and S. Reiss. Linear programming is log space hard for p. *Information Processing Letters*, 8:96–97, 1979.
3. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.
4. R.M. Karp, E. Upfal, and A. Wigderson. The complexity of parallel search. *Journal of Computer and System Sciences*, 36(2):225–253, 1988.
5. J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.
6. M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. In *25th ACM Symp. on Theory of Computing*, pages 448–457, 1993.
7. P.S.Efraimidis and P.G.Spirakis. Positive linear programming extensions: Parallel complexity and applications. Technical Report TR00.06.01, Computer Technology Institute, June 2000.
8. P. Raghavan and C.D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.
9. A.S. Schulz, D.B. Shmoys, and D.P. Williamson. Approximation algorithms. In *Proc. National Academy of Sciences*, volume 94, pages 12734–12735, 1997.

10. M. Serna. Approximating linear programming is log-space complete for p. *Information Processing Letters*, 37(4):233–236, 1991.
11. D. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, A62:461–474, 1993.
12. L. Trevisan and F. Xhafa. The parallel complexity of positive linear programming. *Parallel Processing Letters*, pages 448–457, 1998.

Parallel Shortest Path for Arbitrary Graphs*

Ulrich Meyer and Peter Sanders

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany.
{umeyer,sanders}@mpi-sb.mpg.de.
http://www.mpi-sb.mpg.de/{~umeyer,~sanders}

Abstract. In spite of intensive research, no work-efficient parallel algorithm for the single source shortest path problem is known which works in sublinear time for arbitrary directed graphs with non-negative edge weights. We present an algorithm that improves this situation for graphs where the ratio d_c/Δ between the maximum weight of a shortest path d_c and a "safe step width" Δ is not too large. We show how such a step width can be found efficiently and give several graph classes which meet the above condition, such that our parallel shortest path algorithm runs in sublinear time and uses linear work. The new algorithm is even faster than a previous one which only works for random graphs with random edge weights [10]. On those graphs our new approach is faster by a factor of $\Theta(\log n/\log\log n)$ and achieves an expected time bound of $\mathcal{O}(\log^2 n)$ using linear work.

1 Introduction

The *single source shortest path problem* (SSSP) is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [1]. Let $G = (V, E)$ be a directed graph, $|V| = n$, $|E| = m$, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real-valued *weight* to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the weight of a minimum-weight ("shortest") path from s to v, denoted by *dist(v)*; the weight of a path is the sum of the weights of its edges.

The theoretically most efficient sequential algorithm on directed graphs with non-negative edge weights is Dijkstra's algorithm [5]. Using Fibonacci heaps its running time is given by $\mathcal{O}(n \log n + m)$. Dijkstra's algorithm maintains a partition of V into *settled*, *queued* and *unreached* nodes and for each node v a *tentative distance* $\mathrm{tent}(v)$; $\mathrm{tent}(v)$ is always the weight of some path from s to v and hence an upper bound on $\mathrm{dist}(v)$. For unreached nodes, $\mathrm{tent}(v) = \infty$. Initially, s is queued, $\mathrm{tent}(s) = 0$, and all other nodes are unreached. In each iteration, the queued node v with smallest tentative distance is selected and declared settled and all edges (v, w) are relaxed, i.e., $\mathrm{tent}(w)$ is set to

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 461–470, 2000.
© Springer-Verlag Berlin Heidelberg 2000

$\min\{\text{tent}(w), \text{tent}(v) + c(v,w)\}$. If w was unreached, it is now queued. It is well known that $\text{tent}(v) = \text{dist}(v)$, when v is selected from the queue.

The only known $\mathcal{O}(n \log n + m)$ work parallel SSSP approach for arbitrary directed graphs based on Dijkstra's algorithm uses parallel relaxation of the edges leaving a single node [7]. It has running time $\mathcal{O}(n \log n)$ on a PRAM[1]. All existing algorithms with sublinear execution time require $\Omega(n \log n + m)$ work (e.g., $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3(\log\log n/\log n)^{1/3})$ work [8]). Some less inefficient algorithms are known for planar digraphs [15] or graphs with separator decomposition [3].

Higher parallelism than in Dijkstra's approach can be obtained by a version of the Bellman-Ford algorithm [1] which considers all queued nodes with their outgoing edges *in parallel*. However, it may remove nodes v from the queue for which $\text{dist}(v) < \text{tent}(v)$ and hence may have to *reinsert* those nodes until they are finally settled. Reinsertions lead to additional overhead since their outgoing nodes may have to be *rerelaxed*.

The present paper is based on the Δ-stepping algorithm of [10] which is a generalization of Dijkstra and Bellman-Ford: Tentative distances are kept in an array B of *buckets* such that $B[i]$ stores the unordered set $\{v \in V : v \text{ is queued and } \text{tent}(v) \in [i\Delta, (i+1)\Delta)\}$. In each *phase*, the algorithm removes all nodes from the first nonempty bucket and relaxes all *light* edges ($c(e) \leq \Delta$) of these nodes. This may cause *reinsertions* into the current bucket. For the remaining *heavy* edges, it is sufficient to relax them once and for all when a bucket finally remains empty (see Figure 1). The parameter Δ should be small enough to keep the number of reinsertions small yet large enough to exhibit a useful amount of parallelism.

1.1 Overview and Summary of New Results

The simple parallelization of the Δ-stepping in [10] relies on the particular properties of random graphs with random edge weights thus severely limiting its usage. In Section 2, we introduce a parallel Δ-stepping algorithm which works for arbitrary graphs in time $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta \log n)$ and work $\mathcal{O}(m + n_{\Delta+})$ whp[2]. The parameters which depend on the graph class and the step width are explained in Section 1.2. A further acceleration is achieved in Section 3 by actively introducing *shortcut edges* into the graph thereby reducing the number of times each bucket is emptied to at most two, i.e., the fastest efficient parallel execution time is now $\mathcal{O}((l_\Delta + d_c/\Delta)\log n)$ while performing $\mathcal{O}(m + n'_{\Delta+})$ work whp. In Section 4 it is explained how a good value for the step width Δ (which limits $n'_{\Delta+}$ to $\mathcal{O}(m)$) can be determined efficiently and in parallel. Many of the PRAM results can be adapted to distributed memory machines using techniques described in Section 5. Finally, in Section 6 we summarize the results and apply them on different

[1] We use the *arbitrary CRCW PRAM* model (concurrent read concurrent write parallel random access machine) [9] which specifies that an adversary can choose which access out of a set of conflicting write accesses is successful.

[2] A result holds *with high probability* (*whp*) in the sense that the respective bound is met with probability at least $1 - n^{-\beta}$ for any constant $\beta > 0$.

```
for each v ∈ V do tent(v) := ∞
relax(s, 0);                                          (* Source node at distance 0 *)
while ¬isEmpty(B) do                                  (* Some queued nodes left *)
    i := min{j > i : B[j] ≠ ∅}                        (* Smallest nonempty bucket *)
    R := ∅                                            (* No nodes deleted for bucket B[i] yet *)
    while B[i] ≠ ∅ do                                 (* New phase *)
        Req := findRequests(B[i], light)              (* This may reinsert nodes *)
        R := R ∪ B[i]; B[i] := ∅                      (* Remember deleted nodes *)
        relaxRequests(Req)
    Req := findRequests(R, heavy)                     (* This may reinsert nodes *)
    relaxRequests(Req)

Function findRequests(V′, kind : {light, heavy}) : set of Request
    return {(w, tent(v) + c(v, w)) : v ∈ V′ ∧ (v, w) ∈ E_kind)}

Procedure relaxRequests(Req) for each (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                                 (* Shorter path to w? *)
    if x < tent(w) then                               (* Yes: decrease-key or insert *)
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}         (* Remove if present *)
        B[⌊x/Δ⌋] := B[⌊x/Δ⌋] ∪ {w}
        tent(w) := x
```

Fig. 1. Sequential Δ-stepping.

graph classes. Although our new algorithm is more general than the specialized previous algorithm [10], it turns out to be a factor of $\Theta(\log n / \log\log n)$ faster on random graphs. It has execution time $\mathcal{O}(\log^2 n)$ using linear work.

1.2 Notation and Basic Facts

We have already used d_c as an abbreviation for the maximum weight of a shortest path, i.e., $d_c := \max\{\mathrm{dist}(v) : \mathrm{dist}(v) < \infty\}$. Call an edge disjoint path with weight at most Δ a *Δ-path*. Let C_Δ denote the set of all node pairs $\langle u, v\rangle$ connected by some Δ-path $(u, \ldots, v)$ and let $n_\Delta := |C_\Delta|$. Similarly, define $C_{\Delta+}$ as the set of triples $\langle u, v', v\rangle$ such that $\langle u, v'\rangle \in C_\Delta$ and (v', v) is a light edge and let $n_{\Delta+} := |C_{\Delta+}|$. Let n'_Δ ($n'_{\Delta+}$) denote the number of *simple* Δ-paths (plus a light edge). To simplify notation, we exclude very extreme graphs and assume $n = \mathcal{O}(m)$, $n_\Delta = \mathcal{O}(n_{\Delta+})$ and $n'_\Delta = \mathcal{O}(n'_{\Delta+})$. The *maximum Δ-distance* l_Δ is defined to just exceed the number of edges needed to connect any pair $\langle u, v\rangle \in C_\Delta$ by a path of minimum weight, i.e.,

$$l_\Delta = 1 + \max_{\langle u,v\rangle \in C_\Delta} \min\{|A| : A = (u, \ldots, v) \text{ is a minimum-weight } \Delta\text{-path}\} \ .$$

Similarly, let l'_Δ denote the number of edges in the longest simple Δ-path.

The graph theoretic results from [10] are relatively easy to generalize to see that the number of phases performed by Δ-stepping is bounded by $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta)$ and that the number of reinsertions (rerelaxations) is at most n_Δ ($n_{\Delta+}$). For details refer to the full paper [11] which is available electronically.

2 Parallelization

In this section we develop a first parallelization of Δ-stepping which works for arbitrary graphs and prove the following bound:

Theorem 1. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum path weight d_c, maximum Δ-distance l_Δ and $n_{\Delta+}$ defined as in Section 1.2 can be solved on a CRCW PRAM in time $\mathcal{O}(\frac{d_c}{\Delta} l_\Delta \log n)$ and work $\mathcal{O}(m + n_{\Delta+})$ whp.*

Initialization, loop control, deleting nodes and generating a set 'Req' of node-distance pairs to be relaxed (we call these *requests*) are easy to do in parallel if the nodes are randomly assigned to PUs and if a global array stores the assignment.

The most difficult part is to schedule PUs for actually performing the requests: several relaxations can occur for one node in a phase, and the number of such conflicting relaxations can vary arbitrarily and in an unpredictable way. On CRCW-PRAMs, we can do the PU scheduling efficiently by grouping the requests according to the addressed nodes using the following lemma:

Lemma 1. Semi-sorting *k records with integer keys, i.e., permuting them into an array of size k such that all records with equal key form a consecutive block, can be performed in time $\mathcal{O}(k/p + \log n)$ using p PUs of a CRCW-PRAM whp.*

Proof. First find a perfect hash function $h : V \to 1..ck$ for an appropriate constant c. Using the algorithm of Bast and Hagerup [2] this can be done in time $\mathcal{O}(k/p + \log n)$ (and even faster) whp. Subsequently, we apply a fast, work efficient sorting algorithm for small integer keys such as the one by Rajasekaran and Reif [13] to sort by the hash values.

Once the set of requests 'Req' is grouped by receiving nodes w, we can use prefix sums to schedule $\lfloor p\,|\mathrm{Req}(w)|\,/\,|\mathrm{Req}|\rfloor$ PUs for blocks of size at least $|\mathrm{Req}|\,/p$, and to assign smaller groups with a total of up to $|\mathrm{Req}|\,/p$ requests to individual PUs. The PUs concerned with a group collectively find a request with minimum distance in time $\mathcal{O}(|\mathrm{Req}|\,/p + \log p)$ and then relax it in constant time.

Summing the work and time for all $l_\Delta d_c/\Delta$ phases yields the desired bound.

3 Finding Shortcuts

In the analysis of the number of phases for our algorithms we bounded the maximum number of iterations, l_Δ, that are required until the current bucket under consideration remains finally empty. It was already noticed in [6] that only one iteration per bucket is needed if the bucket width is smaller than any edge weight. No reinsertions occur in that case but in the presence of very small edge weights, the number of buckets, d_c/Δ, might become very large due to the small Δ. However, l_Δ can be reduced to 2 by explicitly introducing a *shortcut edge* (u, v) for each node pair connected by a Δ-path.

What interests us here is how to find these edges in parallel and how the search itself can be performed in a load balanced way. Although, we do not know a general algorithm doing that using $\mathcal{O}(m + n_{\Delta+})$ work, we can solve the problem if the number of simple Δ-paths is not too large. More precisely, the remainder of this section is devoted to establishing the following Theorem:

Theorem 2. *There is an algorithm which inserts an edge* (u, v) *with weight* $c(u, v) = \mathrm{dist}(u, v)$ *for each shortest path* $(u, \ldots, v)$ *with* $\mathrm{dist}(u, v) \leq \Delta$ *using* $\mathcal{O}(l'_\Delta \log n)$ *time and* $\mathcal{O}(m + n'_{\Delta+})$ *work on a CRCW PRAM whp. (Where* $\mathrm{dist}(u, v)$ *denotes the weight of a shortest path from* u *to* v*.)*

Applying the results from Section 2 we get:

Corollary 1. *The single source shortest path problem for directed graphs with* n *nodes,* m *edges, maximum path weight* d_c *and* $n'_{\Delta+}$, l'_Δ *as defined in Section 1.2 can be solved on a CRCW PRAM in time* $\mathcal{O}((l'_\Delta + \frac{d_c}{\Delta}) \log n)$ *and work* $\mathcal{O}(m + n'_{\Delta+})$ *whp.*

Figure 2 outlines a routine which finds shortcuts by applying a variant of the Bellman-Ford algorithm to all nodes in parallel. It solves an all-to-all shortest path problem constrained to Δ-paths. The shortest connections found so far are kept in a hash table of size $\mathcal{O}(n'_{\Delta+})$ (we can use dynamic hashing if we do not know a good bound for $n'_{\Delta+}$). This table plays a role analogous to that of $\mathrm{tent}(\cdot)$ in the main routine of Δ-stepping. The set Q stores *active* connections, i.e., triples (u, v, y) where y is the weight of a shortest known path from u to v and where paths $(u, \ldots, v, w)$ have not yet been considered as possible shortest connections from u to w with weight $y + c(v, w)$. In iteration i of the main loop, the shortest connections using i edges are computed and are then used to update 'found'. Applying similar techniques as before, this routine can be implemented to run in $\mathcal{O}(l'_\Delta \log n)$ parallel time using $\mathcal{O}(m + n'_{\Delta+})$ work: We need l'_Δ iterations each of which takes time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|Q'|)$ whp. The overall work bound holds since for each simple Δ-path $(u, \ldots, v)$, $\langle u, v\rangle$ can be a member of Q only once. Hence, $\sum_i |Q| \leq n + n'_\Delta$ and $\sum_i |Q'| \leq n + n'_{\Delta+}$.

```
Function findShortcuts(Δ) : set of weighted edge
    found : HashArray[V × V]                          (* return ∞ for undefined entries *)
    Q := {(u, u, 0) : u ∈ V}                          (* (start, destination, weight) *)
    Q' : MultiSet
    while Q ≠ ∅ do
        Q' := ∅
        for each (u, v, x) ∈ Q dopar
            for each light edge (v, w) ∈ E dopar
                Q' := Q' ∪ {(u, w, x + c(v, w))}
        semi-sort Q' by common start and destination node
        Q := {(u, v, x) : x = min{y : (u, v, y) ∈ Q'}}
        Q := {(u, v, x) ∈ Q : x ≤ Δ ∧ x < found[(u, v)]}
        for each (u, v, x) ∈ Q dopar found[(u, v)] := x
    return {(u, v, x) : found[(u, v)] < ∞}
```

Fig. 2. CRCW-PRAM routine for finding shortcut edges

4 Determining Δ

In the case of arbitrary edge weights it is necessary to find a step width Δ which is large enough to allow for sufficient parallelism and small enough to keep the

algorithm work-efficient. Although we expect that application specific heuristics can often give us a good guess for Δ relatively easily, for a theoretically satisfying result we would like to be able to find a good Δ systematically.

We now explain how this can be done if the adjacency lists have been preprocessed to be *partially sorted*: Let $\Delta_0 := \min_{e \in E} c(e)$ and assume[3] that $\Delta_0 > 0$. The adjacency lists are organized into *blocks* of edges with weight $2^j \Delta_0 \leq c(e) < 2^{j+1} \Delta_0$ for some integer j. Blocks with smaller edges precede blocks with larger edges.[4]

Theorem 3. *Let n'_Δ, $n'_{\Delta+}$ and l'_Δ be defined as in Section 1.2 and consider an input with partially sorted adjacency lists. For any constant α, there is an algorithm which identifies a step width Δ, such that $n'_{\Delta+} \leq \alpha m$ and $n'_{2\Delta+} > \alpha m$, and which can be implemented to run in $\mathcal{O}((l'_\Delta + \log \frac{\Delta}{\Delta_0}) \log n)$ time using $\mathcal{O}(m)$ work whp.*

The basic idea is to reuse the procedure findShortcuts(Δ) of Figure 2 but to divide the computation into *rounds*. In round i, $0 \leq i \leq \log \frac{\max_{e \in E} c(e)}{\Delta_0}$), we set $\Delta_{\text{cur}} = 2^i \Delta_0$ and find all connections (u, v, x) with $\Delta_{\text{cur}} \leq x < 2\Delta_{\text{cur}}$. In order to remain work efficient, a number of additional measures are necessary however. We now outline the changes compared to the routine 'findShortcuts' from Figure 2. Most importantly, we have a bucketed *todo-list* T. $T[i]$ stores entries (u, v, x, b) where (u, v, x) stands for a connection from u to v with weight x, and b points to the first block in the adjacency list of v which may contain edges (v, w) with $2^i \Delta_0 \leq x + c(v, w) < 2 \cdot 2^i \Delta_0$. (Note that the number of buckets may be arbitrarily large. In this case, we store the buckets in a dynamic hash table and only initialize those buckets which actually store elements.)

At the beginning of round i, for each entry (u, v, x, b) of $T[i]$, the adjacency list of v is scanned beginning at block b until a block is encountered which cannot produce any candidate connections for bucket i. A new entry of the todo list is produced for the first bucket $k > i$ for which it can produce candidate connections. The candidate connections found are used to initialize Q'.

Both this initialization step and the iteration on Q can produce candidate connections whose weights reach into bucket $i+1$. After removing duplicates and longer connections than found before, we therefore split the remaining candidates into the new content of Q and a set Q_{next} storing connections with weight in bucket $i + 1$.

At the end of round i, when Q finally remains empty, we create new entries in the todo-lists for all connections newly encountered in round i. In order to do that, we keep track of all new entries into 'found' using two sets S and S_{next} for connections with weights in bucket i and $i + 1$, respectively. Q_{next} and S_{next} are used to initialize Q and S in the next round respectively.

[3] This assumption can be removed.

[4] This preprocessing is trivially parallelizable on a node-by-node basis, we get a good parallel preprocessing algorithm for the case $p = \mathcal{O}(n/d)$ if d is the maximum out-degree of a node.

The total number of connection-edge pairs considered is monitored so that the whole procedure can be stopped as soon as it is noticed that this figure exceeds αm. At this time, the entries of 'found' constitute at least all simple $(\Delta_{\text{cur}}/2)$-paths. Thus, taking $\Delta := \Delta_{\text{cur}}/2$ as the final step width, it is guaranteed that the number of reinsertions and rerelaxations in a subsequent application of the Δ-stepping will be bounded by $\mathcal{O}(m)$. On the other hand, $n'_{2\Delta+} > \alpha m$.

Using an analogous analysis as for the function 'findShortcuts' it turns out that the search for Δ can be implemented to run in $\mathcal{O}((l'_\Delta + \log \frac{\Delta}{\Delta_0}) \log n)$ time using $\mathcal{O}(m)$ work where l'_Δ denotes the number of edges in the longest simple Δ-path.

5 Adaptation to Distributed Memory Machines

In this section we consider the following distributed memory model: There are p processing units (PUs) numbered 0 through $p-1$ which are connected by a communication network. Let $T_{\texttt{routing}}(k)$ denote the time required to route k constant size messages per PU to random destinations. Let $T_{\texttt{coll}}(k)$ bound the time to perform a (possibly segmented) reduction or broadcast involving a message of length k and assume that $T_{\texttt{coll}}(x)+T_{\texttt{coll}}(y) \leq T_{\texttt{coll}}(1)+T_{\texttt{coll}}(x+y)$, i.e., concentrating message length does not decrease execution time. Note, that on powerful interconnection networks like multiported hypercubes we can achieve a time $\mathcal{O}(\log p + k)$ whp for $T_{\texttt{routing}}(k)$ and $T_{\texttt{coll}}(k)$.

So far it is unknown how to efficiently implement the linear work semi-sorting procedure for load-balancing on distributed memory[5]. However, if shortcuts are present we now explain how this problem can be circumvented. We also assume that the nodes can be randomly assigned to PUs using a constant time hash function[6] $\text{ind}(\cdot)$ and that we know $\text{indegree}(v)$ when looking at an edge (u, v).

Theorem 4. *Given a directed graph G with n nodes, m edges, maximum path weight d_c and $n_{\Delta+}$, l_Δ as defined in Section 1.2. Under the assumptions given above, the single source shortest path problem can be solved in time*

$$\mathcal{O}\left(\overline{m} + T_{\texttt{routing}}(\overline{m}) + T_{\texttt{coll}}(\overline{m}) + \frac{d_c}{\Delta}(T_{\texttt{coll}}(1) + T_{\texttt{routing}}(1))\right)$$

on a distributed memory machine with p PUs for $\overline{m} = \frac{m+n_{\Delta+}}{p}$ and any given source node s whp.

We first simplify the search algorithm to exploit the fact that in the presence of shortcuts, classifying edges as light or heavy is no longer important for the

[5] The preprocessing can be done (somewhat inefficiently) by implementing semi-sorting using ordinary sorting or using a slower yet work efficient algorithm requiring $\mathcal{O}(T_{\texttt{routing}}(n^\epsilon))$ time for any positive constant ϵ. Both alternatives yield a work-efficient algorithm for powerful interconnection networks if the preprocessing overhead can be amortized over sufficiently many source nodes.

[6] This is a common assumption, e.g., in efficient PRAM simulation algorithms.

shortest path search itself. By explicitly treating intra-bucket edges (source and target reside in the same bucket) first, each edge is relaxed at most once: After buckets 0 through $i-1$ have been emptied, a single relaxation pass through the edges reaching from $B[i]$ into $B[i]$ suffices to settle all nodes now in $B[i]$. After that, $B[i]$ can be emptied by relaxing all edges reaching out of $B[i]$ once.

The two most difficult parts are (1) generating the set of requests, i.e. identifying the set of edges that are to be relaxed and (2) assigning the requests to their nodes and scheduling the PUs for performing the relaxations. We start with (1):

In a distributed memory setting we cannot dynamically schedule outgoing edges between the PUs in the same way as we did for PRAMs. Scanning adjacency lists to generate requests is therefore load balanced using a static assignment of edges to PUs: An adjacency list of size outdegree(v) is collectively handled by an *out-group* of PUs. Out-groups are selected as follows: W.l.o.g., assume that p is a power of two minus one and the PUs are logically arranged as a complete binary tree. If outdegree(v) $> p$ then all PUs participate in v's out-group. Otherwise, a subtree rooted at a random PU is chosen which is just large enough to accommodate one edge per PU, i.e., it contains $2^{\lceil \log(\text{outdegree}(v)+1)\rceil}-1$ nodes. Requests for a bucket can now be generated by first sending the tentative distance of the nodes in $B[i]$ to the roots of out-groups responsible for them. (We will later see where this information comes from.) Then, the PUs pass all the node-distance pairs they have received down the tree in a pipelined fashion and do the same for the distances of the nodes received from above.

Now consider a fixed leaf PU j for a fixed iteration of the algorithm. (Since interior tree-nodes pass all their work downwards, interior PUs have no more work to do than a leaf node.) Let $X_i := 1$ if PU j is part of the out-group of a node i expanded in this iteration and $X_i = 0$ otherwise. We have $\mathbf{P}[X_i = 1] = 2^{-h(i)}$ if the root of the out-group of node i is $h(i)$ levels away from the root of the PU-tree. The total number of nodes PU j has to work on is $Y := \sum_{i=1}^{k} X_i$ if k is the number of nodes expanded in the current iteration and $\mathbf{E}[Y] = \sum_i 2^{-h(i)}$. By definition of the size of subtrees, we get $\mathbf{E}[Y] = \mathcal{O}(K/p)$ if K is the total number of edges leaving nodes expanded in this iteration. Using a Chernoff bound with nonuniform probabilities [12, Theorem 4.1], it is now easy to see that $Y = \mathcal{O}(K/p + \log n)$ whp. Since the communication pattern is just a slightly generalized form of a broadcast, distributing the tentative distances can be done in time $\mathcal{O}(T_{\texttt{coll}}(K/p + \log n))$ whp. Summing over all iterations we get time $\mathcal{O}(T_{\texttt{coll}}(m/p + \log n) + T_{\texttt{coll}}(1)d_c/\Delta)$. Generating the actual request values is then possible using local computations only.

Now we tackle problem (2): how to assign the requests to nodes and schedule PUs for performing the relaxations. The idea for arbitrary graphs is to postpone the relaxation of an edge until the latest possible moment – just before the bucket of the target node is emptied. Since edges are relaxed only once (recall that we assume the presence of short-cuts), it pays to allocate an *in-group* of size $2^{\lceil \log(\text{indegree}(v)+1)\rceil} - 1$ for node v analogously to the way out-groups are allocated. Each PU maintains an additional bucket structure B_q for the nodes

for which it is part of the in-group. Requests are routed to a preassigned position in the in-group, but this information is only used to place the node into B_q. So, after iteration $i-1$ is computed, the content of $B[i]$ is not yet known. Rather, we first have to find $B[i] = \bigcup B_q[i]$. This can be done locally for each in-group using a pipelined tree operation which is the converse of the operation used for broadcasting in the out-groups. (Each PU maintains a hash table of nodes already passed up the tree.) Then, the result is broadcast to all PUs in the in-groups so that from now on, redundant entries of nodes in buckets beyond $B[i]$ can be deleted. Also, edges which have not received a request yet are marked as superfluous. Requests ending up there in later iterations will simply be discarded. Finally, the actual global minima are computed using another pipelined reduction operation. Now, the heads of the in-groups are ready to send the tentative distances of nodes in $B[i]$ to the heads of the out-groups. The analysis of these tree-operations is analogous the analysis for the out-groups.

6 Conclusion

The parameters governing the performance of Δ-stepping are the maximum path weight d_c and the largest step width Δ which ensures that there is only a linear number of Δ-connections (plus a light edge), n_Δ ($n_{\Delta+}$). If we want to introduce shortcuts efficiently, the choice of Δ must also bound the number of simple Δ-paths (plus a light edge), n'_Δ ($n'_{\Delta+}$). For parallelization, the corresponding l'_Δ has some influence too: On a CRCW PRAM our new algorithm with shortcut insertion needs $\mathcal{O}((l'_\Delta + \frac{d_c}{\Delta}) \log n)$ time and $\mathcal{O}(m + n'_{\Delta+})$ work whp.

We now instantiate the result for some input graph classes. As a role model we look at general graphs with maximum in-degree and out-degree d and *random edge weights*, uniformly distributed[7] in the interval $[0,1]$. For $\Delta = \Theta(1/d)$ we have $l'_\Delta = \mathcal{O}(\log n / \log\log n)$ whp and $\mathbf{E}[n_{\Delta+}] \leq \mathbf{E}[|P_{2\Delta}|] = \mathcal{O}(n)$ [10]. Thus, we get expected parallel time $\mathcal{O}((dd_c + \log n)\log n)$ and linear work. For example, for r-dimensional meshes with random edge weights we have $d_c = \mathcal{O}(n^{1/r})$ and hence execution time $\mathcal{O}(n^{1/r} \log n)$ using linear work for any constant r.

For random graphs from $\mathcal{G}(n, d/n)$, i.e., with edge probability d/n and random edge weights the maximum path weight is $d_c = \mathcal{O}(\log n/d)$ whp [10]. Thus, with our new approach we get an $\mathcal{O}(\log^2 n)$ parallel time linear expected work PRAM algorithm. This is a factor $\Theta(\log n / \log\log n)$ better than the best previously known work efficient algorithm from our earlier paper [10].

Another example are random geometric graphs $G_n(r)$ where n nodes are randomly placed in a unit square and each edge weight equals the Euclidean distances between the two involved nodes. An edge (u,v) is included if the Euclidean distance between u and v does not exceed the parameter $r \in [0,1]$. Random geometric graphs have been intensively studied since they are considered to be a relevant abstraction for many real world situations [14, 4]. Taking $r = \Theta(\sqrt{\log(n)/n})$ results in a connected graph with $m = \Theta(n \log n)$ edges and

[7] The results carry over to some other random distributions, too.

$d_c = \mathcal{O}(1)$ whp. For $\Delta = r$ the graph already comprises all relevant Δ-shortcuts such that we do not have to explicitly insert them. Consequently our PRAM algorithm runs in $\mathcal{O}((1/r)\log n)$ parallel time and performs $\mathcal{O}(n+m)$ work whp.

Acknowledgements

We would like to thank in particular Hannah Bast, Kurt Mehlhorn and Volker Priebe for many fruitful discussions and suggestions. Hannah Bast also pointed out the elegant solution of using a perfect hash function for semi-sorting requests.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows : theory, algorithms and applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *3rd Symposium on Parallel Algorithms and Architectures*, pages 50–61, 1991.

[3] Edith Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, September 1996.

[4] J. Diaz, J. Petit, and M. Serna. Random geometric problems on $[0,1]^2$. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518, pages 294–306. Springer, 1998.

[5] E.W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[6] E. A. Dinic. Economical algorithms for finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44, 1978.

[7] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31, 1988.

[8] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 353–362, San Diego, CA, USA, June 1992. ACM Press.

[9] Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, 1992.

[10] U. Meyer and P. Sanders. Δ-stepping: A parallel shortest path algorithm. In *6th European Symposium on Algorithms (ESA)*, number 1461 in LNCS, pages 393–404. Springer, 1998.

[11] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. `http://www.mpi-sb.mpg.de/~sanders/papers/long-delta.ps.gz`, 1999.

[12] J. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[13] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[14] R. Sedgewick and J. S. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1:31–48, 1986.

[15] Jesper Larsson Träff and Christos D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel algorithms for irregularly structured problems : Intern. workshop (IRREGULAR-3)*, volume LNCS 1117, pages 183–194S., Berlin, 1996. Springer.

Periodic Correction Networks*

Marcin Kik

Institute of Computer Science, Wrocław University
kik@tcs.uni.wroc.pl

Abstract. We present a comparator network that sorts sequences obtained by changing a limited number of keys in a sorted sequence. The network that we present is periodic and has a depth 8. The time required by this algorithm is $O(\log n+k)$ with a small constant hidden by "Oh" notation (n is the total number of keys, k is the number of changed keys).

1 Introduction

Sorting on comparator networks. Sorting is one of the fundamental computer science problems both in theory and practice. One of the models for designing sorting algorithms are comparator networks. We are given a set of registers $\{R_0, R_1, \ldots, R_{n-1}\}$, each register capable of holding a single key. The goal is to sort these keys, i.e. to relocate them so that their ordering agrees with the ordering of registers. For this purpose, we apply *compare-exchange operations*. Such an operation compares numbers x, y stored in R_i and R_j and stores $\min\{x, y\}$ in R_i and $\max\{x, y\}$ in R_j. In the situation described, we say also that there is a comparator (R_i, R_j) between R_i and R_j that performs the compare-exchange operation. Comparators might be executed in parallel provided that no register is involved in more than one operation at a time. Such a collection of comparators is called a *layer*. *Comparator network* is an algorithm which executes a fixed set of comparators (independent of data). The comparators of a comparator network are organized into several layers; the number of these layers is called *depth* of the network and corresponds to the execution time. The input keys are placed in the registers of the network, the ith key in register R_i. Once a sorting algorithm terminates its execution, the keys stored in $R_0, R_1, \ldots, R_{n-1}$ form a nondecreasing sequence.

There are comparator networks that sort n keys in $O(\log n)$ parallel steps [1] (with $\log n$ being an obvious lower bound for the number of parallel steps). However, they are not practical because of a big constant hidden by the big "Oh" notation.

Correcting disturbed sequences. One of the problems arising in practical computing is re-sorting data that once has been sorted, but a limited number of keys has been changed. If the number of such keys is k, we say that the sequence is k-*disturbed*. Of course, for sorting k-disturbed sequences one may apply general sorting algorithms, but this might be less efficient than using specially tailored methods.

* partially supported by Komitet Badań Naukowych, grant 8 T11C 032 15

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 471–478, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The problem considered in this paper is how to sort efficiently k-disturbed sequences on comparator networks for k substantially smaller than n. We call such networks *k-correction networks*. Some partial solutions to this problem are known: Schimmler and Starke [8] present a network with $O(n)$ comparators correcting 1-disturbed sequences in time $2\log n - 1$. A network of depth $4\log n + O(\log^2 k \log\log n)$ that sorts k-disturbed sequences for $k \leq n$ is presented in [4]. Some further results have been reported by G. Stachowiak.

Periodic comparator networks. A *periodic network* processes the input sequence in many iterations, i.e. the output of the network is processed in the next iteration as the input. Thus, the network may have even a constant depth and still be capable of sorting arbitrary sequences. A motivation for periodic comparator networks is to reduce amount of hardware necessary to implement a sorting algorithm.

For the sorting problem, many periodic comparator networks have been designed. The first significant step was a periodic network of depth $\log n$ sorting in $\log n$ iterations [3]. Later, quite fast networks of a constant depth have been invented for many architectures ([9], [5], [6]).

In this paper we present periodic comparator network of a constant depth capable of sorting disturbed sequences:

Theorem 1. *There is a periodic comparator network of depth* 8 *that sorts* k*-disturbed sequences in time* $O(\log n + k)$.

2 Preliminaries

One of the fundamental observations concerning comparator networks is that a comparator network N sorts all sequences if and only if N sorts all zero-one sequences (i.e. sequences of zeroes and ones) ([7], page 224). This principle let us restrict the analysis of a network to zero-one inputs only. The same phenomenon holds for k-disturbed sequences, in order to prove that a network sorts all k-disturbed sequences it suffices to show it for sorted zero-one sequences in which at most k keys have changed the value from zero to one or from one to zero.

For a zero-one sequence with x zeroes the *zeroes area* (respectively, *ones area*) is the set of registers R_0 through R_{x-1} (respectively, R_x through R_{n-1}). A *displaced* key is a one in the zeros area or a zero in the ones area. It is easy to show (see [4]) that any k-disturbed zero-one sequence contains at most k displaced zeroes and at most k displaced ones.

The simplest periodic sorting network is *odd-even transposition network*. Its first layer contains comparators (R_i, R_{i+1}), for even i, the second layer consists of comparators (R_i, R_{i+1}), where i is odd. This is an extremely simple architecture, however the sorting time is n.

A common strategy in analysis of a comparator network is to isolate a *dirty area*, that is a set of registers $R_j, \ldots, R_h$, such that $R_0, \ldots R_{j-1}$ contain only zeroes and $R_{h+1}, \ldots R_{n-1}$ contain only ones. Then the *clean* areas remain clean and we may restrict our attention to comparators with both endpoints in the dirty area.

It is relatively easy to design a periodic 1-correction network with runtime $O(\log n)$. The problem that we have to solve is that displaced elements may block each other while going into proper positions. This phenomenon known for arbitrary correction networks becomes a nasty one for periodic correction networks. Due to the periodic nature of the algorithm we cannot adjust the steps in order to respond appropriately to problems arising at different phases of sorting.

3 Periodic k-Correction Network

De nition of the Network. In this section we present the network $P_{l,w'}$ ($l > 1$ and $w' > 1$ are parameters of the construction) with the properties stated in Theorem 1, for $w' = 6k$. Let us remark at this point that some features of $P_{l,w'}$ are introduced just to enable the analysis, and may be unnecessary for a good performance in practice.

Let $w = 2(l + 1 + w')$, $h = 2^l$. The input size of $P_{l,w'}$ is $n = wh$. Let $R = \{R_0, \ldots, R_{n-1}\}$ be a set of registers. We define a function $r : \{0, \ldots, w-1\} \times \{0, \ldots, h-1\} \to R$ by $r(x, y) = R_{x+wy}$. We arrange the registers of $P_{l,w'}$ in a matrix, where the register $r(x, y)$ is placed in column x and row y. (The row numbers increase downwards, i.e. row y is above the row $y + 1$.)

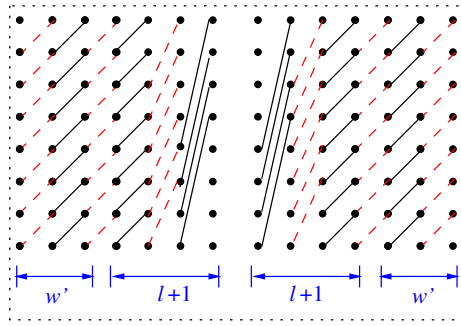


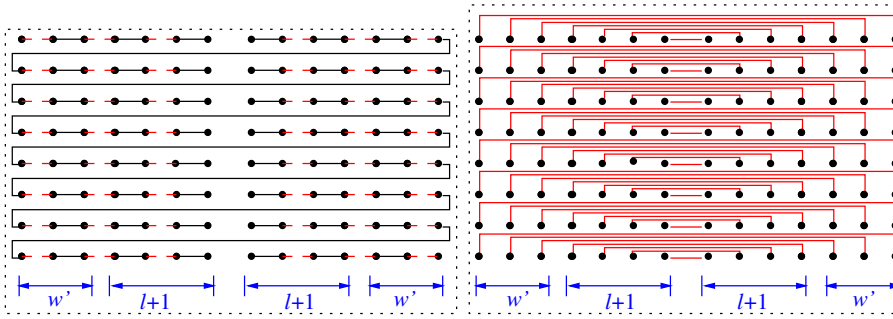
Fig. 1. Back-jump layers M_1 (solid lines) and M_3 (dashed lines).

The network $P_{l,w'}$ has 8 layers $(M_0, M', M_1, M', M_2, M', M_3, M')$. First we define M_1 and M_3 (called *back-jump* layers): Let $C_x = \{(r(x + 1 + \frac{w}{2}, y), r(x + \frac{w}{2}, y + 2^{\max\{0,l-1-x\}})) \in R^2\}$. Let C'_x be the set of comparators (R_{n-1-t}, R_{n-1-s}) such that (R_s, R_t) is in C_x. M_1 (respectively, M_3) is the union of the sets $C_x \cup C'_x$, for even (respectively, odd) x, $0 \le x < \frac{w}{2} - 1$ (see Fig. 1).

M_0 and M_2 (called *horizontal* layers) are defined as follows (see Fig. 2 (a)). Let $D_x = \{(R_s, R_{s+1}) \in R^2 \mid s \bmod w = x\}$. Then M_0 (respectively, M_2) is the union of D_x, for $0 \le x \le w - 1$, where $x - \frac{w}{2}$ is even (respectively, odd) and $x \neq \frac{w}{2} - 1$.

We also define a *left-right* layer M' as $\{(r(x, y), r(w - 1 - x, y)) \mid 0 \le x < \frac{w}{2}\}$ (see Fig. 2 (b)).

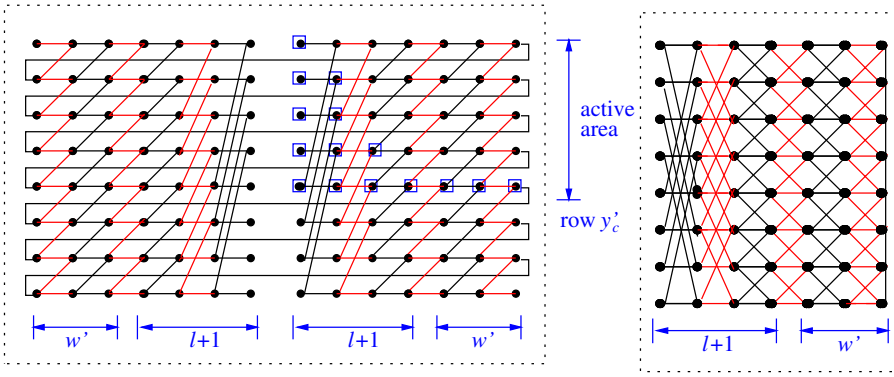
We partition the set of registers R into the *left set* $S_0 = \{r(x, y) \mid 0 \le x < \frac{w}{2}\}$ and the *right set* $S_1 = R \setminus S_0$. The members of S_0 (respectively, S_1) are called



(a) (b)

Fig. 2. Layers M_0 (solid lines) and M_2 (dashed lines) (a), and M' (b).

left (respectively, *right*) registers. Fig. 3(b) presents $P_{3,3}$ in a *folded* state (i.e. with S_0 rotated 180 degrees around the central vertical axis). For any register $r = r(x, y)$, we define a *shadow* of r, denoted by $\mathrm{s}(r)$, as $r(w - 1 - x, y)$. For $X \subseteq R$, $\mathrm{s}(X) = \{\mathrm{s}(r) \mid r \in X\}$. Note that in the folded state, each register is in the same place as its shadow, and that M' contains comparators of the form $(\mathrm{s}(r), r)$, for $r \in S_1$.



(a) (b)

Fig. 3. $P_{3,3}$ without the left-right comparators (a), and folded $P_{3,3}$ (b).

Sketch of the Runtime Analysis of $P_{l,w'}$. Let $L = (L_0, \ldots, L_7)$ be the sequence of layers of $P_{l,w'}$ (i.e. $L_{2i+1} = M'$ and $L_{2i} = M_i$). Let c be an arbitrary zero-one k-disturbed sequence. We show that $P_{l,w'}$ sorts c in $O(w' + l) = O(k + \log n)$ iterations. Let y'_c be the index of the first row of registers that intersects the ones area. For $t \geq 0$, let c_t be a sequence obtained after execution of t steps (i.e layers) of the iterated $P_{l,w'}$

on c. We use $c(r)$ to denote the r-th element of the sequence c (i.e. the value stored in register R_r). We apply a technique that is reverse to zero-one principle in some sense. Let c' be a sequence defined as follows.

$$c'(r) = \begin{cases} 0 & \text{if } c(r) = 0, \\ \sum_{p=0}^{r} c(p) & \text{if } c(r) = 1 \text{ and } R_r \text{ is above row } y'_c, \\ k+1 & \text{if } c(r) = 1 \text{ and } R_r \text{ is below row } y'_c - 1. \end{cases}$$

The sequence c has at most k displaced ones. Thus the registers above the row y'_c, which contain displaced ones in c, contain the values from the range $\{1, \ldots, k\}$ in c', every value occurring exactly once. The sequences c'_t and c''_t are defined as follows:

- $c''_0 = c'$.
- For $t \geq 0$, c'_t is the sequence c''_t with all the values from the range $\{1, \ldots, k\}$ below the row $y'_c - 1$ replaced by $k + 1$.
- For $t \geq 0$, c''_{t+1} is the output of $L_{t \bmod 8}$ applied to c'_t.

Note that $c_t(r) = 1$ if and only if $c'_t(r) > 0$. We will show that for $t = O(l + w')$ there are only zeroes above the row $y'_c - 1$ in c'_t (and hence in c_t).

Active area and stoppers. Below, we start to investigate in detail the fine structure of $P_{l,w'}$. We call the set of registers above the row y'_c an *active area* and denote it by S' (see Fig. 3 (a)). The registers of S' are called *active*. Let $S'_0 = S_0 \cap S'$ and $S'_1 = S_1 \cap S'$. We call $r \in S'_1$ a *stopper* if and only if there is no back-jump comparator of the form (r, r') such that r' is in the active area. The stoppers on Fig. 3 are marked with the boxes. By *active values* we mean the values from the range $\{1, \ldots, k\}$ stored in the active registers. Note that any active value may disappear (be replaced by $k + 1$) if it is compared with a zero from outside S'.

Zones and the levels of registers. We partition the set S'_1 into zones $Z_{i,j}$ and $Z'_{i,j}$ defined as follows (see Fig. 4 (a)):

- for $0 \leq x \leq l - 1$, $Z_{x,0}$ (respectively, $Z'_{x,0}$) is the set of all stoppers $r(\frac{w}{2} + x, y)$ such that $y'_c - y > 2^{l-1-x}$ (respectively, $y'_c - y \leq 2^{l-1-x}$).
- for $l \leq x \leq \frac{w}{2} - 1$, $Z_{x,0} = \{r(\frac{w}{2} + x, y'_c - 1)\}$,
- for $x' > 0$, $Z_{x,x'}$ (respectively, $Z'_{x,x'}$) is the set of $r \in S'_1$ such that there is a back-jump comparator (r, r') with $r' \in Z_{x,x'-1}$ (respectively, $r' \in Z'_{x,x'-1}$).

It is easier to analyze the movements of active values between the zones than their detailed routes. A *zones tree* T is a directed graph with the nonempty zones as vertices and the set of arcs $E = E_1 \cup E_2$ (see arrows on Fig. 4 (b)), where E_1 contains the arcs of the form $(Z_{x,x'+1}, Z_{x,x'})$ and $(Z'_{x,x'+1}, Z'_{x,x'})$ (*back-jump arcs*) and $E_2 = E_{2,1} \cup E_{2,2} \cup E_{2,3}$ (*horizontal arcs*), where $E_{2,1} = \{(Z_{x,0}, Z'_{x,1}) \mid 0 \leq x \leq l - 1\}$, and $E_{2,2} = \{(Z'_{x,0}, Z_{x+1,0}) \mid 0 \leq x \leq l - 1\}$, and $E_{2,3} = \{(Z_{x,0}, Z_{x+1,0}) \mid l \leq x \leq \frac{w}{2}\}$. The *root* of T is $Z_{\frac{w}{2}-1,0}$. For each zone Z we define its *level* (denoted by $\mathrm{l}(Z)$) as its distance from the root of T, and, for each $r \in Z \cup \mathrm{s}(Z)$, $\mathrm{l}(r) = \mathrm{l}(Z)$. For each active value i, let *level of* i in c'_t be the level of active register that contains i or zero if i does not exist in c'_t. The levels of the zones are displayed on Fig. 4 (b). Let l'_T denote the maximal level of a zone in T. By considering the path from arbitrary vertex of T to the root, it is easy to note that l'_T is $O(w' + l)$.

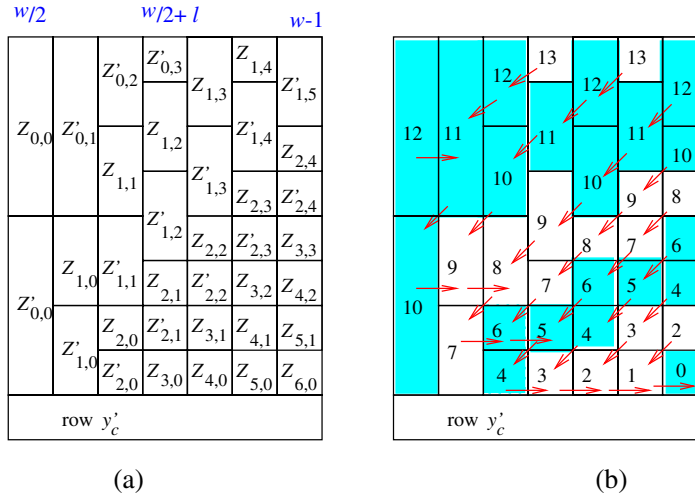


Fig. 4. Partition of the right active registers into the zones (a) and the levels of the zones (b). The arrows on (b) represent the arcs of the tree of zones T.

Releasing the registers from the active values. For each $t \geq 0$, for active i, we say that a register R_r *is released* from i at step t if and only if for each $t' \geq t$, $c'_{t'}(r) \neq i$.

Consider the following simple example: Suppose we have the odd-even transposition network with an input consisting of the k positive values: $1, \ldots, k$, placed in arbitrary registers and zeroes placed in all the remaining registers. Note that after the first computation step, the register R_0 is released from k. After the second step the registers R_0 and R_1 are released from k and thus after the third step the register R_0 is released from $k-1$. In such a way we can define, for each $t \geq 0$ and $i > 0$, the set of registers that are released from i at step t. Note that the border of the area released from $i-1$ is adjacent to the border of the area released from i.

We apply analogous reasoning. For each $l \geq 0$, we define A_l as $\{r \in S'_1 \mid \mathrm{l}(r) \leq l\}$. Note that $A_{l'_T} = S'_1$ and $A_0 = Z_{\frac{w}{2}-1,0}$. We partition the set of levels into *groups of levels*. Note that for $0 < x < l$, we have $\mathrm{l}(Z'_{x-1,0}) - \mathrm{l}(Z'_{x,0}) = 3$. Let $b = \mathrm{l}(Z'_{0,0}) \bmod 3$. We define the ith *group of levels* as $\mathcal{G}_i = \{j \mid 3i + b \leq j \leq 3i + b + 2\}$. The zones with the levels in the odd and even groups have been depicted by different shades on Fig. 4 (b). By the definition of b, the level of each zone $Z'_{x,0}$ is a minimum of some group of levels. The *first phase* of the computation consists of $6l'_T$ iterations.

Lemma 1. *For each active value i, $S' \setminus (A_{\max \mathcal{G}_{2(k-i)}} \cup \mathrm{s}(A_{\min \mathcal{G}_{2(k-i)}}))$ is released from i after $6l'_T$ iterations.*

Sketch of the proof. The proof is based on the following claims.

Claim. For each active value i, if $S' \setminus (A_{\max \mathcal{G}_j} \cup \mathrm{s}(A_{\min \mathcal{G}_j}))$ is released from the values greater than i at step t, and i is inside $A_{\min \mathcal{G}_{j+1}} \cup \mathrm{s}(A_{\max \mathcal{G}_j})$ in sequence c'_t, then $S' \setminus (A_{\max \mathcal{G}_{j+1}} \cup \mathrm{s}(A_{\min \mathcal{G}_{j+1}}))$ is released from i at step t.

There is no greater value than i in $S' \setminus (A_{\max \mathcal{G}_j} \cup \mathrm{s}(A_{\min \mathcal{G}_j}))$. Thus i can leave $\mathrm{s}(A_{\max \mathcal{G}_j})$ by a left-right comparator or being pulled back by a greater value directly to $A_{\max \mathcal{G}_j}$, or by going forward through a horizontal or back-jump comparator to a shadow of a zone with level $\min \mathcal{G}_{j+1}$. In the last case i is moved by the next left-right layer to $A_{\min \mathcal{G}_{j+1}}$. i can leave $A_{\min \mathcal{G}_{j+1}}$ by entering some zone with level $\min \mathcal{G}_{j+1}+1$ or by going from the column $w-1$ to 0. In the first case i is moved by the next back-jump layer back to $A_{\min \mathcal{G}_{j+1}}$. In the second case i either enters $\mathrm{s}(A_{\max \mathcal{G}_j})$ or is moved by the next left-right layer back to $A_{\min \mathcal{G}_{j+1}}$.

Claim. For each active value i, if $S' \setminus (A_{\max \mathcal{G}_j} \cup \mathrm{s}(A_{\min \mathcal{G}_j}))$ is released from the values greater than i after iteration t (i.e. at step $8t$), and $S' \setminus (A_{\max \mathcal{G}_{j+2}} \cup \mathrm{s}(A_{\min \mathcal{G}_{j+2}}))$ is released from i after iteration t, then within the next 6 iterations i must visit some register from $A_{\min \mathcal{G}_{j+1}} \cup \mathrm{s}(A_{\max \mathcal{G}_j})$ thus releasing $S' \setminus (A_{\max \mathcal{G}_{j+1}} \cup \mathrm{s}(A_{\min \mathcal{G}_{j+1}}))$.

We skip the details of the proof. There are no greater values than i in $S' \setminus (A_{\max \mathcal{G}_j} \cup \mathrm{s}(A_{\min \mathcal{G}_j}))$. Thus while i is outside $A_{\min \mathcal{G}_{j+1}} \cup \mathrm{s}(A_{\max \mathcal{G}_j})$ it must enter $A_{\max \mathcal{G}_{j+2}}$ by a left-right comparator, and then:

- move through the back-jump comparators until it reaches a stopper,
- move through the horizontal and back-jump comparators according to the arcs of T (or by a horizontal comparator directly from a lower half of some $Z'_{x,0}$ to $Z'_{x+1,0}$).

During each of the above steps, level of i is decreased by at least 1 and each iteration contains at least one such step.

By the last claim we have: for $t \geq 0$, for each active i, $S' \setminus (A_{l_{t,i}} \cup \mathrm{s}(A_{l_{t,i}-2}))$ is released from i after the iteration $6 \cdot t$, where $l_{t,i} = \max \mathcal{G}_{\max\{2(k-i), l'_T - t + 2(k-i)\}}$.

Thus after the first phase all active values will have the levels not greater than $\max \mathcal{G}_{2(k-1)} = 6k - 4 + b$. Since $b \leq 2$ and $w' = 6k$, they will remain in $A_{w'-1} \cup \mathrm{s}(A_{w'-1})$. This part of a network has a very simple structure. It can be shown that after $O(w')$ iterations of the next (*second*) phase $S' \setminus (A_{k-i} \cup \mathrm{s}(A_{k-i}))$ is released from i, for each active value i. We add one more iteration to the second phase to ensure that both k and $k-1$ are moved to A_0 and A_1 respectively.

Final smoothing of displaced elements. After the second phase each active i is inside $A = A_{k-i} \cup \mathrm{s}(A_{k-i})$. The final goal is to move them into the last row of A_k, call it B, and its shadow, B'. Now we change our conventions: every nonzero element below active area and in B is replaced by $k+1$ as soon as it arrives there.

To each register r in $A \setminus B$ we assign a label $q_t(r)$ that increases during the computation so that $k - q_t(r)$ is an upper bound on the active value that can still appear in r. Initially, B' is released from k and $k-1$ so initial labels (i.e. q_0) of its registers are 1.5. The registers in $A \setminus (B \cup B')$ have initial labels equal to their levels. For $r \in A \setminus B$, if there is a comparator inside $(A_k \setminus B) \cup B'$, of the form (r, r') or $(\mathrm{s}(r), r')$ such that $q_t(r) - q_t(r') = 0.5$, then all the registers with the label $q_t(r)$ are connected in a similar way to some register with the label $q_t(r) - 0.5$ and we set $q_{t+1}(r) = q_t(r) + 0.5$. (Note that either $q_t(r)$ is an integer and the register r' is released from $k - q(r) + 1$ and $k - q_t(r)$ can move from r to r' in at most single iteration, or we can increase $q_t(r)$ by 0.5 without destroying the bound on the active values that can be in r.) If there is no such comparator, then $q_{t+1}(r) = q_t(r)$.

It is straightforward to show that all registers in $A \setminus (B \cup B')$ have the label q_{2k} greater than k. Hence, after $2k$ iterations of the third phase all the active values are in the row $y'_c - 1$.

We have shown that $P_{l,w'}$ moves all displaced ones below the row $y'_c - 2$ and (by symmetry) all displayed zeroes above the row $y'_c + 2$ in at most $O(l + w')$ iterations. The rows y'_{c-1}, y'_c and y'_{c+1} are then sorted in $O(w)$ iterations, by the comparators of the odd-even transposition network contained in the horizontal and left-right layers.

4 Conclusions

In this paper, we have shown that k-disturbed sequences may be corrected efficiently by constant depth periodic networks. However, our solution remains efficient for small k's only (i.e. $k = o(\log^3 n)$), since for $k = \Omega(\log^3 n)$ the "periodification" [6] of a Batcher sorting network [2] is more efficient than our solution. It is an open question how to obtain periodic constant depth correction networks with runtime $c \log n + o(k)$.

Acknowledgment. Thanks to Grzegorz Stachowiak for helpful discussions and to Krzysztof Loryś for comments on this paper.

References

1. M. Ajtai, J. Komlós and E. Szemerédi. Sorting in $c \log n$ parallel steps. Combinatorica, Vol. 3, pages 1–19, 1983.
2. K. E. Batcher. Sorting networks and their applications. Proceedings of 32nd AFIPS, pages 307–314, 1968.
3. M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The periodic balanced sorting network. Journal of the ACM, Vol. 36, pages 738–757, 1989.
4. M. Kik, M. Kutyłowski and M. Piotrów. Correction networks. Proceedings of the IEEE-ICPP, pages 40-47, 1999.
5. M. Kik, M. Kutyłowski and G. Stachowiak. Periodic constant depth sorting network. Proceedings of the 11th STACS, pages 201–212, 1994.
6. M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, and R. Wanka. Fast and feasible periodic sorting networks. Proceedings of the 55th IEEE-FOCS, 1994. Full version to appear in the Journal of ACM.
7. D. E. Knuth. The art of Computer Programming. Volume 3: Sorting and Searching. Addison-Wesley, 1973.
8. M. Schimmler, C. Starke. A Correction Network for N-Sorters. SIAM Journal on Computing, Vol. 6, No. 6, 1989.
9. U. Schwiegelshohn. A short-periodic two-dimensional systolic sorting algorithm. IEEE International Conference on Systolic Arrays, pages 257–264, 1988.

Topic 07
Applications on High-Performance Computers

Michael Resch

Local Chair

Parallel computing - similarily to all scientific research areas - is torn between two opposites brilliantly described by Max Weber [1] and Jose Ortega y Gasset [2]. Weber advocated the approach of specialization by diving into small details of a problem as deeply as possible while Ortega y Gasset favoured an encyclopedic approach of combining knowledge of different fields.

Several chapters of these proceedings deal with very detailed aspects of parallel computing. The authors follow Max Weber's recommendations and focus on a single aspect only to get as much into detail as possible to get to the bottom of the problem. The emphasis of the chapter here is on applied sciences and the authors are following the more encyclopedic approach of Jose Ortega y Gasset.

The papers clearly show the fast development of parallel computing within the last years. A number of standards have been defined and a number of tools have been provided. MPI and OpenMP have helped to overcome the Babylonian confusion of languages. All this has contributed to establishing parallel computing as a tool for computational scientists. The few papers presented here can certainly not give a complete overview of the range of applications that have benefitted from this process, but they can highlight some of the most interesting developments.

The topic is split into two sections. The first section is more devoted to applications that make use of parallel computing. The second section is focusing on tools that support the development of applications.

R. E. Lynch, H. Lin and Dan C. Marinescu are working with electronic micrographs. They have an interest in the reconstruction of asymmetric objects which can be performed on clusters of PCs. This once again shows the potential of such systems. Sergio Romero, Luis F. Romero and Emilio L. Zapata present a topic that is more unusual for computer scientists. Their aim is to simulate the behavior of clothes which is a complex task requiring both sheer performance and sophisticated load balancing. Gordon J. Darling, Terence M. Sloan and Connor Mulholland are interested in Geographic Information Systems. Input, handling and processing of vector-topological data has turned out to be a difficult task. Some optimization must be done in order to fully benefit from parallel computing. El Mostafa Daoudi, Abdelouafi Meziane and Yahya Ould Mohamed El Hadj study load balancing which is necessary for automatic speech recognition. Using a Hidden Markov Model (HMM), they show how to improve existing methods.

Piotr Bala and Terry W. Clark elaborate on Pfortran and Co-Array Fortran. Although standards like MPI and OpenMP are already well established, the authors expound the potential that innovative approaches still posess. Markus

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 479–480, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Ast, Cristina Barrado, Jose M. Cela, Rolf Fischer, Jesus Labarta, Oscar Laborda, Hartmut Manz and Uwe Schulz focus on the tricky problem of solving sparse matrix problems. By introducing an approach based on dynamic scheduling, they can achieve excellent speedups for real applications. Ken Naono, Yusaku Yamamoto, Mitsuyoshi Igai, Hiroyuki Hiramaya and Nobuhiro Ioki describe an implementation of a real symmetric eigensolver on parallel processors. In order to achieve better performance in the inverse iteration part, they introduce a multi-color framework where the inverse iterations are executed concurrently. Very good results can be demonstrated on the most recent SR8000 system. Andrei Jalba and Felicia Ionescu investigate the efficient parallelization of Fast Hartley Transforms (FHT) on shared memory multiprocessors.

References

1. Max Weber, 'Vom inneren Beruf zur Wissenschaft', in Max Weber 'Soziologie, Universialgeschichtliche Analyse, Politik', Kröner, Stuttgart, 1992.
2. Jose Ortega y Gasset, 'Der Aufstand der Massen', Deutsche Verlagsanstalt, Stuttgart.

An Efficient Algorithm for Parallel 3D Reconstruction of Asymmetric Objects from Electron Micrographs*

Robert E. Lynch, Hong Lin, and Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{rel, linh, dcm}@cs.purdue.edu,
http://www.cs.purdue.edu/homes/sb/Projects/P3DR/P3DR.html

Abstract. Recently we proposed a 3D reconstruction algorithm based on Fourier analysis using Cartesian coordinates. In this algorithm the computations to determine the values of the 3D Discrete Fourier Transform of the density of an asymmetric object could be naturally distributed over the nodes of a parallel system. Now we present an improvement of this algorithm which, for reconstruction at points of an $N \times N \times N$ grid, uses $O(N^3)$ arithmetic operations instead of $O(N^5)$. The algorithm is general and can be used for 3D reconstruction of asymmetric objects for applications other than structural biology.

1 Introduction

There are several practical methods for reconstructing a 3D object from a set of its 2D projections. These include use of Fourier Transforms, back projection, and numerical inversion of the Radon Transform. See [9] for a review of these and other methods. Also, for descriptions of (sequential) methods for 3D reconstruction and related tasks, see [7], [8], and [10], three of several books containing clear explanations and many references.

Thirty years ago, Crowther et al., [6], presented several Fourier methods for 3D reconstruction. One of them, based on Fourier-Bessel Transforms, has been used extensively for symmetric objects by the structural biology community. In [13] an outline is given of our first parallel algorithm for 3D reconstruction which was based on a another Fourier method proposed in [6] that does not require symmetry. Here we present an improvement which, for reconstruction at points of an $N \times N \times N$ grid, uses $O(N^3)$ arithmetic operations instead of $O(N^5)$ as in [13].

The amount of experimental data needed for reconstruction of an asymmetric object is considerably larger than for a symmetric one. A typical reconstruction

* The research reported in this paper was partially supported by grants from National Science Foundation, MCB 9527131 and DBI-9986316.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 481–490, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of an icosahedral virus at, say, 20Å resolution might require a few hundreds projections, e.g., 300, each of which is used 60 times, whereas the reconstruction of an asymmetric object with the same dimensions and at the same resolution would require 60 times more data, i.e., 18,000 projections. The amount of experimental data also increases when reconstruction of larger virus-antibody complexes is attempted. It is not unrealistic to expect an increase, of three to four orders of magnitude, in the volume of experimental data for high resolution asymmetric objects.

X-ray crystallography is the only method to obtain high resolution (2–2.5Å) electron density maps for large macromolecules like viruses, while until recently electron microscopy was only able to provide low resolution (20Å) maps. Cryo-`EM` is appealing to structural biologists because crystallizing a virus is sometimes impossible and even when possible, it is technically more difficult than preparing samples for microscopy. Thus the desire to increase the resolution of cyo-`EM` methods to the 5Å range. In the last years results in the 7–7.5Å range have been reported, [3], [5]. But increasing the resolution of the `3D` reconstruction process requires more experimental data. It is estimated that the number of views to obtain high resolution electron density maps from cryo-`EM` micrographs chould increase by two orders of magnitudes, from the current hundreds to tens of thousands.

Even though nowadays fast processors and large amounts of primary and secondary storage are available at a relatively low cost, the `3D` reconstruction of asymmetric objects at high resolution requires computing resources, CPU cycles, primary and secondary storage, well beyond those provided by a single system. Thus the need for parallel algorithms.

To use efficiently a parallel computer or a cluster of workstations, we need parallel algorithms that partition the data and computations evenly among nodes to ensure load balance and, moreover, which minimize the communication among processors by maintaining a high level of locality of reference. Similar efforts have been reported in the past [11], but the performance data available to us suggest that new algorithms have to be designed to reduce dramatically the computation time.

The atomic structure determination of macromolecules based on electron microscopy is an important application of `3D` reconstruction. The procedure for structure determination consists of the following steps:

Step A Extract individual particle projections from micrographs and identify the center of each projection.
Step B Determine the orientation of each projection.
Step C Carry out the `3D` reconstruction of the biological macromolecule.
Step D Dock an atomic model into the `3D` density map.

The development of parallel algorithms to carry out some of these computations is part of an ambitious effort to design an environment for 'real-time electron microscopy', where results can be obtained in hours or days rather than in weeks or months [16].

Algorithms for automatic identification of particle projections, the determination of the center and orientation of each projection (Steps A and B) are discussed in [15], and parallel algorithms to determine the orientation are presented in [2]. In this paper we are only concerned with Step C, which is carried out as follows:

Step 1 Compute the 2D Discrete Fourier Transform (DFT) of each projection.
Step 2 Use interpolation and least squares to compute an estimate of the 3D DFT of the electron density.
Step 3 Compute the inverse 3D DFT to get an estimate of the electron density.

Some of these computations can be done independently from each other. For example in Step 1 each processor can be assigned a set of projections and carry out the 2D DFT concurrently. Data exchange among nodes is necessary to collect information for Step 2 and then each node calculates the Fourier Coefficients on its own set of 3D planes. Different portions of the 3D DFT of the electron density are stored on different nodes; where possible, we carry out 2D inverse transforms and then data are exchanged among nodes and so that the final set of 1D inverse transforms takes place to complete Step 3.

2 3D Reconstruction by Fourier Transforms

We outline the relationship between the experimentally observed projections and the electron density, ρ, of the macromolecule. For more details, see, for example, [6], [8], or [17].

Suppose that the macromolecule is centered at the origin and is inside a cube having side length A. The Fourier Series representation of ρ is

$$\rho(\mathbf{x}) = \sum_{\mathbf{h}} F(\mathbf{h}) e^{2\pi i \mathbf{h}^T \mathbf{x}/A}$$

where $\mathbf{x}^T = (x, y, z)$, $\mathbf{h}^T = (h, k, \ell)$, and ^{T} denotes the transpose of a vector; here $\mathbf{h}$ has integer components. Because the density is zero outside the cube, F is also the Fourier Transform of ρ:

$$F(\mathbf{h}) = \frac{1}{A^3} \int \rho(\mathbf{x})\, e^{-2\pi i\, \mathbf{h}^T \mathbf{x}/A}\, d\mathbf{x},$$

and this applies for $\mathbf{h}$ having integer or noninteger components.

The 'Projection Theorem' states that the 3D Fourier Transform of ρ at points on a plane through the origin is equal to the 2D Fourier Transform of the projection of ρ onto that same plane. This result is immediate for the case that $\mathbf{h}^T = (h, k, 0)$ and follows because the Fourier Transform of a function after a rotation about the origin is the same as the rotation of the original transformation.

A point (u, v) on the plane of projection is also a point $\mathbf{h}' = (h', k', \ell')^T$ in the h, k, ℓ coordinate system. Consequently, the value of $F(\mathbf{h}')$ is the Fourier

Transform, $P(u,v)$, of the projection of the density onto the (u,v)-plane. Hence we have

(1)

$$F(\mathbf{h}') = \frac{1}{A^3}\int\left\{\sum_{\mathbf{h}} F(\mathbf{h})e^{2\pi i\mathbf{h}^T\mathbf{x}/A}\right\} e^{-2\pi i\,\mathbf{h}'\mathbf{x}/A}\,d\mathbf{x} = \sum_{\mathbf{h}} F(\mathbf{h})\frac{1}{A^3}\int e^{2\pi i(\mathbf{h}-\mathbf{h}')^T\mathbf{x}/A}\,d\mathbf{x}$$

$$= P(u,v) = \sum_{h,k,\ell} F(h,k,\ell)\,\mathrm{sinc}(h-h')\,\mathrm{sinc}(k-k')\,\mathrm{sinc}(\ell-\ell'),$$

where

$$\mathrm{sinc}(s) = \left\{\begin{matrix}\sin(\pi s)/\pi s & \text{if } s\ /=0\\ 1 & \text{if } s=0\end{matrix}\right\} = \frac{1}{A}\int_{-A/2}^{A/2} e^{2\pi i\,s\,t/A}\,dt.$$

Piecewise Constant Model. Variation of the value inside a pixel square cannot be measured, and thus we take the measured projection p to be a piecewise constant function on the pixel frame:

$$p_{\mathbf{t}}(r,s) = p_{\mathbf{t}}(i\Delta r, j\Delta s) \quad \text{for } |r-i\Delta r| < \Delta r/2,\ \ |s-j\Delta s| < \Delta s/2,\ \text{with } \Delta r = \Delta s,$$

where i and j are integers, the subscript $\mathbf{t}$ denotes a unit vector normal to the plane of the pixel frame, and $(i\Delta r, j\Delta s)$ denotes grid points at the center of pixel squares.

If we regard this function to be defined on a *plane*, then we are lead to the system in (1) which relates the measured values $P(u,v)$ (the 2D transform of p) to the unknown values $F(h,k,\ell)$ of the 3D transform of ρ. Except at the origin, it is unlikely that any of the (h,k,l) grid points would be be in the plane of the projection of a randomly oriented molecule; similarly in the transform space.

But, if we regard the projection as a function defined on a *slab*, of thickness Δr, instead of on a *plane*, then this not only gives a formulation which is consistent in the three coordinate directions, but also leads to an efficient algorithm for determining the $F(h,k,l)$.

We now extend each pixel value to be a constant in a cube having edge-length equal to the side-length of a pixel. Similarly, the value $P(u,v)$ of the transformed pixel is extended to a constant on a cube in transform space. Figure 1 indicates this extension for the simpler 2D case.

When such a cube contains the 3D grid point (h,k,ℓ), then we set

$$P(u,v) = F(h,k,\ell)\,\mathrm{sinc}(h-h')\,\mathrm{sinc}(k-k')\,\mathrm{sinc}(\ell-\ell') + G(u,v;h,k,\ell),$$

where G is the error. Minimization of the sum of the squares of all the G's associated with a given grid point (h,k,ℓ) (i.e., set the derivative of $\sum G^2$ equal to zero) yields

$$F(h,k,\ell) = \left\{\sum_{m,n} P_m(u_{m,n},v_{m,n})\,S_{m,n}\right\} \Big/ \left\{\sum_{m,n} S_{m,n}^2\right\} \tag{2}$$

where

$$S_{m,n} = \sum_{m,n} \mathrm{sinc}(h-h'_{m,n})\mathrm{sinc}(k-k'_{m,n})\mathrm{sinc}(\ell-\ell'_{m,n}),$$

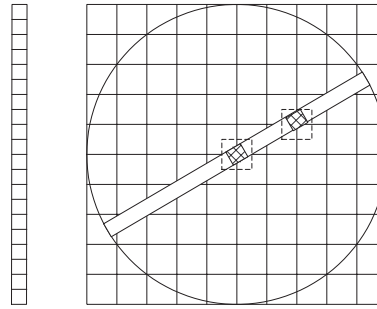


Figure 1. Slab projection simplified to the 2D case. A 1D pixel frame (a line segment) extended to a 2D 'strip' is indicated on the left. The figure on the right shows this strip in its correct orientation on the 2D domain of the Discrete Fourier Transform. When a 2D grid point is in the strip, as indicated by a shaded square, the corresponding 1D transform value P is taken as the 2D transform value F at the grid point in the corresponding square with dashed boundary.

where P_m denotes the a m-th transformed pixel frame, and $(u_{m,n}, v_{m,n})$ denotes a grid point in the m-th frame whose associated mesh-cube contains (h, k, ℓ). Use of (2) requires two orders of magnitude less arithmetic than the method discussed in [13].

The Effect of Zero-Fill. One can put a pixel frame into a larger array and fill the extra array entries will zeros. One can use this "zero-fill" to try to improve accuracy.

As shown in [4] (p. 90 ff), the DFT of a zero-filled array gives an interpolant to the transform of the $P \times P$ pixel values on a finer grid. For example, when $k = 2$ and $f_{j,k}$ and $F_{\ell,m}$ denote DFT values on the $P \times P$ and $2P \times 2P$ arrays, respectively, then $f_{j,k} = KF_{2j,2k}$, where K is a known constant that depends on the normalization used in the DFT.

Although this does not give an interpolant to the transform of the projected electron density, it does give an interpolant on a finer grid to the observed pixel values. Having values on a finer grid, one obtains greater accuracy when interpolating to the pixel values.

3 Results of Numerical Experiments

To assess the accuracy of the algorithm, we reconstructed the density of a uniform sphere. We used $P \times P$ pixel frames, having P pixels on each side. The pixel values were unity inside a circle of diameter D pixels, and zero outside the circle. Before computing the Discrete Fourier Transform, the $P \times P$ pixel frame was put into a $kP \times kP$ array, with $k \geq 1$. The extra array entries were set equal to zero; we call this "zero-fill" and k the "aspect ratio". The results we reporte elsewhere [14] indicate that embedding the pixel frames into larger arrays, decreases the numerical errors in the reconstruction process but it does

increases the amount of space and the number of arithmetic operations. For for the uniform sphere test case, the error inside the sphere decreased from about 25% for a zero-fill aspect ratio of $k = 1$ to less than 4.5% for $k = 4$. We also report some results for similar problems obtained with the sequential program, `EM3DR` [1], used by the some structural biologists.

Random numbers uniformly distributed between −5% and +5% of the maximum projected value were added to the pixel frames of projections of the uniform sphere to simulate the effect of the noise on the reconstruction process. Reconstructed densities indicate that the noise leads to noticeable distortion of the sphere.

4 Performance of This Parallel Program

A program implementing the algorithm presented in this paper was written and tested using projections of a uniform sphere as well as several experimental data sets. The program is written in Fortran, uses the MPI library for communication, and was designed to run efficiently on a cluster of inexpensive PCs. We use a cluster of 16, 400 MHz Pentium II processors running SunOS5.6. Each processor has 256 MB of main memory and a 8 GB disk. The connectivity is provided by a 100 MBps Ethernet switch. The total cost of the system is about $40K. The actual performance of this system is comparable for this problem with the performance of a 16 processor SGI Origin 2000.

The program is based on a data parallel execution model, all nodes perform essentially the same computation but on different data. The coordinator node reads the input files containing the set of projections and the orientation of each projection and then distributes the projections evenly among the set of available nodes. Then, each node processes the individual frames assigned to it by doing an `2D` DFT; if the aspect-ratio $k > 1$, it first extends the pixel array with zero fill. A data exchange stage occurs at the end of the Fourier analysis phase, each node is assigned a set of linear equations. After solving the linear systems the nodes carry out a `2D` DFT then a global exchange takes place and a `1D` FFT completes the Fourier synthesis phase. Finally, the coordinator node gets individual `2D` sections of the `3D` map from the other nodes and writes the electron density map out. We use this method because we do not have a parallel file system and several nodes reading the input data concurrently and then writing the output density maps concurrently would lead to an unacceptable performance degradation due to I/O contention.

We are primarily interested in the load balancing properties of the algorithm and in the speedup of the implementation. The tests conducted with the uniform sphere gave us confidence in the correctness of the algorithm and its implementation. We then used actual data collected in cryo-`EM` experiments as indicated in Table 1, to make additional test of the correctness of our program. We used only data for symmetric objects because our objective was to compare our results with the results produced by the sequential program `EM3DR`.

Problem	Virus	Pixels	Views	Symmetry
A	Polyomavirus (Papovaviruses)	69 × 69	158 × 60	Icosahedral
B	Papillomavirus (Papovaviruses)	99 × 99	60 × 60	Icosahedral
C	Sindbis virus (Alphaviruses)	221 × 221	389 × 60	Icosahedral
D	Sindbis virus (Alphaviruses)	221 × 221	643 × 60	Icosahedral
E	Paramecium Bursaria Chlorella Virus, type 1	281 × 281	107 × 60	Icosahedral
F	Ross River Virus (Alphaviruses)	131 × 131	1777 × 10	Dihedral
G	Bacteriophage Phi29	191 × 191	609 × 10	Dihedral
H	Auravirus (Alphaviruses)	331 × 331	1940 × 60	Icosahedral
I	Paramecium Bursaria Chlorella Virus, type 1	359 × 359	948 × 60	Icosahedral

Table 1. Data for 9 problems used to test the parallel 3D reconstruction program. For symmetry S (60 or 10), P projections give $P \times S$ views.

We profiled the program to determine the time used for each execution phase. Table 2 shows that interpolation is the most intensive phase, followed by the 2D Fourier analysis, while solving the linear systems requires a relatively small amount of arithmetic operations. In [13] we reported that solving the linear systems was the most time-consuming phase of 3D reconstruction (with our previous algorithm).

Execution Phase\Problem	A	B	C
Initialization	0.26	0.13	0.13
2D Fourier Analysis	1.75	1.2	75.9
Interpolation	11.49	8.34	270.2
Data Exchange for Solvesys	0.0016	0.006	0.073
Solvesys and Combine	0.019	0.053	0.64
2D Fourier Synthesis	0.067	0.23	4.49
Data Exchange for 1D Synthesis	0.010	0.027	0.33
1D Fourier Synthesis	0.030	0.10	2.15
Gather Data	0.0050	0.015	0.18
Write Density Map	0.2	0.45	3.11

Table 2. Time (in seconds) for each phase of the parallel program, when run on a single node, for problems A, B and C.

Table 3 shows the time used in each node when the program solves one of the problems in multiple nodes. From the data in Table 3, one sees that the computation is evenly distributed among multiple nodes. The coordinator (Node 1) carries out extra processing in the initial and final phases of the execution.

Table 4 shows the seedups for the nine problems presented above. The reduction in speedup when Problems A and B were run on 8 nodes was due to the relatively large amount of time devoted to data communication, synchronization, etc., for these small problems. Because of the sizes of problems H and I, we were unable to run in one node and report only the speedups relative to the running time in two nodes.

Node:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	567.2															
2	322.5	318.9														
4	162.0	158.8	158.8	158.8												
8	91.7	88.7	88.7	88.7	88.7	88.7	88.7	88.7								
16	60.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6	57.6

Table 3. Time (in seconds) by each node for Problem D. Execution with 1, 2 ,4, 8, and 16 nodes.

Node Number \ Problem	A	B	C	D	E	F	G	H	I
2	1.82	1.85	1.82	1.76	1.73	1.94	1.92		
4	2.79	2.86	3.57	3.50	3.25	3.63	3.21	1.93×2	1.87×2
8	1.16	1.54	6.18	6.19	4.31	6.32	5.77	4.04×2	2.77×2
16			8.59	9.35	5.98	7.73	7.00	7.62×2	4.84×2

Table 4. Speedups of the parallel program. Problems H and I required at least 2 nodes.

5 Summary

The algorithm discussed in this paper is general and can be used for `3D` reconstruction of asymmetric objects from `2D` projections in applications other than structural biology.

Our experimental results indicate that embedding the pixel frames into larger arrays, a technique we call "zero-fill", decreases the numerical errors in the reconstruction process but it does increases the amount of space and the number of arithmetic operations. For for the uniform sphere test case, the error inside the sphere decreased from about 25% for a zero-fill aspect ratio of $k = 1$ to less than 4.5% for $k = 4$. The magnitude of the mean square errors of a `3D` reconstruction program based on the algorithm presented in this paper is slightly lower than the ones for the sequential program, `EM3DR`, based on the algorithm described in [6] when the zero-fill aspect ratio is one and significantly lower when we increase k.

In practice, the data collected in cryo-electron microscopy is subject to experimental errors due to various factors, e.g., variations of the intensity of the beam, the non-uniform layer of ice, and other sources of noise. Additional errors occur when extracting the individual projections from the micrographs, determining the center of each projection, etc. The traditional wisdom is that if the number of projections that is used is much larger than the minimum number required for reconstruction, then the effect of these errors can be overcome. Indeed many structures have been solved even though the reconstruction was carried out at relatively low resolution. Our results support our intuition that errors have a non-uniform distribution, the farther we are from the center of the sphere the larger are the errors. We studied also the effect of the number of projections on the magnitude of errors. Since we are performing a Monte Carlo calculation, we expect that the mean square error should decrease as $1/\sqrt{\texttt{number of views}}$.

But, the mean square error seem to decrease at a slower rate, e.g., in one case, the error in the density inside a uniform sphere was only 4.59% for 1250, 4.51% for 5000, and 4.45% for 20100 projections. This is probably due to the jump discontinuity at the edge of the sphere.

We discuss the accuracy of the algorithm and note that the results computed by a parallel program implementing the algorithm are consistent with the ones produced by a sequential program EM3DR used for many years for structural biology studies [1]. We report the speedup and the load balancing results for processing cryo-EM data for several viruses. One iteration of the 3D reconstruction for the Bursalia Corella Virus that used to take about 4 hours using EM3DR was carried out in less than 3 minutes on 16 nodes using the program based on our algorithm.

The most time consuming phases of the program execution are: (a) the interpolation, (b) the 2D Fourier analysis, and (c) the initialization phase where input files containing the projections and the orientation of each projection are read in. In our previous experiments [13], we found that the most time consuming phase of the program was solving linear systems. The load balancing results are very good. In most cases the execution times of all but the coordinator node are within 1% of each other. The need to exchange data among nodes and to synchronize after each phase reduces somewhat the speedup on realistic problems. We report on, the results of 3D reconstruction for 8 virus structures. The speedup in 4 nodes is about 3.5, ranges from a low of 3.7 to a high of 6.9 for 8 nodes and is in the range of 7 to 11 for 16 nodes. Some of problems (A and B) might be too small to attain the expected speedup: additional improvements in the implementation of the algorithm are needed.

The experiments reported in this paper were conducted on a low-cost parallel system consisting of a cluster of 16 Pentium II processors running at 400 MHz, each with 256 MB of memory and interconnected by a 100 Mbps Ethernet. With a faster interconnection network our results would be better.

6 Acknowledgments

The authors are grateful for many insightful discussions with Michael G. Rossmann and Timothy S. Baker. Robert Ashmore and Wei Zhang from Baker's lab provided assistance with the sequential reconstruction program EM3DR and test data.

References

1. Baker, T. S., J. Drak, and M. Bina, "Reconstruction of the three-dimensional structure of simian virus 40 and visualization of chromatin core" Proc. Natl. Acad. Sci. USA, 85:422-426, 1988.
2. Baker, T. S., I. M. B. Martin, and D. C. Marinescu, "A parallel algorithm for determining orientations of biological macromolecules imaged by electron microscopy", CSD-TR 97-055, 1997.

3. Böttcher, B., S. A. Wynne, and R. A. Crowther, "Determination of the fold of the core protein of hepatitis B virus by electron cryomicroscopy", Nature (London) 386, 88–91, 1997.
4. Briggs, W. L., and V. E. Henson, *The* `DDT`*, An Owner's Manual for the Discrete Fourier Transform*, SIAM Publications, 1995.
5. Conway, J. F., N. Cheng, A. Zlomick, P. T. Wingfield, S. J. Stahl, and A. C. Steven, "Visualization of a 4-helix bundle in the hepatitis B virus capsid by cryo-electron microscopy". Nature (London) 386, 91–94, 1997.
6. Crowther, R. A., D. J. DeRosier, and A. Klug, "The reconstruction of a three-dimensional structure from projections and its application to electron microscopy", Proc. Roy. Soc. Lond. A 317, 319–340, 1970.
7. Deans, S. R., *The Radon Transform and Some of Its Applications*, 2nd Edit., Krieger Publishing Company, 1993.
8. Frank, J., *Three-Dimensional Electron Microscopy of Macromolecular Assemblies*, Academic Press, 1996.
9. Gordon, R., "Three-dimensional reconstruction from projections: A review of algorithms", Intern. Rev. of Cytology 38, 111–151, 1974.
10. Grangeat, P., and J-L Amans, Eds., *Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, Kluwer Academic Publishers, 1996.
11. Johnson, C. A., N. I. Weisenfeld, B. L. Trus, J. F. Conway, R. L. Martino, and A. C. Steven, "Orientation determination in the 3D reconstruction of icosahedral viruses using a parallel computer", CS & E, 555–559, 1994.
12. Lynch, R. E., and D. C. Marinescu, "Parallel 3D reconstruction of spherical virus particles from digitized images of entire electron micrographs using Cartesian coordinates and Fourier analysis", CSD-TR #97-042, Department of Computer Sciences, Purdue University, 1997.
13. Lynch, R. E., D. C. Marinescu, H. Lin, and T. S. Baker, "Parallel algorithms for 3D reconstruction of asymmetric objects from electron micrographs," Proc. IPPS/SPDP (13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing), pp. 632–637, 1999.
14. Lynch, R. E., H. Lin, and D. C. Marinescu "An algorithm for parallel 3D reconstruction of asymmetric objects from electron micrographs," 1999 (submitted).
15. Martin, I. M., D. C. Marinescu, T. S. Baker, and R. E. Lynch, "Identification of Spherical particles in digitized images of entire micrographs", J. of Structural Biology, 120, 146–157, 1997.
16. Martin, I. M. and D. C. Marinescu "Concurrent Computations and Data Visualization for Spherical Virus Determination", IEEE Computational Science & Enginerering, October-December, pp. 40-51, 1998.
17. Rossmann, M. G., and Y. Tao, "Cryo-electron-microscopy reconstruction of partially symmetric objects", J. Structural Biology, 125, 196–208, 1999.

Fast Cloth Simulation with Parallel Computers

Sergio Romero, Luis F. Romero, and Emilio L. Zapata

Universidad de Málaga, Dept. de Arquitectura de Computadores, P.O. Box 4114, E-29080, Spain,
{sromero,felipe,ezapata}@ac.uma.es

Abstract. The computational requirements of cloth and other non-rigid solid simulations are high and often the running time is far from real time. In this paper, an efficient solution of the problem on parallel computer is presented. An application, which combines data parallelism with task parallelism has been developed, achieving a good load balancing and minimizing the communication cost. The execution time obtained for a typical problem size, its super-linear speed-up, and the iso-scalability shown by the model, will allow to reach real-time simulations in sceneries of growing complexity, using the most powerful multiprocessors.

1 Introduction

Cloth and flexible material simulation is an essential topic in computer animation of realistic virtual humans and dynamic sceneries. New emerging technologies, as interactive digital television and multimedia products, make necessary the development of powerful tools able to perform real time simulations. There are several approaches to simulate flexible materials. These methods can be classified in physically-based, geometrical and hybrid models (a combination of both). The former provide reliable representations of the behavior of the materials, while the others require a high degree of user intervention making them unusable for interacting applications. In this work, a physically-based method has been chosen. In a physical approach, clothes and other non-rigid objects are usually represented by interacting discrete components (finite elements, springs-masses, patches) each one numerically modeled by an ordinary differential equation: $\ddot{x} = M^{-1}f(x, \dot{x})$, where x is the vector of position of the masses M. The derivatives of x are the velocities $\dot{x}$ and the accelerations $\ddot{x}$.

The model presented considers both spring-mass and triangle mesh formulations. The former is usually applied with solids, while the later works better for clothes. Equations in most formulations contain non-linear components, that are typically linearized using a Newton method, generating a linear system of algebraic equations where positions and velocities are the unknowns. So, these different formulations can be merged, giving an unified system where 2D and 3D models are simultaneously solved, an essential topic when interactions among bodies occur. The use of explicit integration methods, such as forward Euler and Runge-Kutta, results in easily programmable code and accurate simulations [6]. They have been broadly used during the last decade, but recent works [1]

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 491–499, 2000.
© Springer-Verlag Berlin Heidelberg 2000

demonstrate that implicit methods overcome the performance of explicit ones, assuming a non visually perceptible lost of precision. In the composition of virtual sceneries, appearance, rather than accuracy, is required. So, in this work, an implicit technique, backward Euler method, has been used to solve equation (1) where $v = \dot{x}$.

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ M^{-1}f(x,v) \end{pmatrix} \tag{1}$$

The detection of collision between simulated objects is critical and the computational cost can be extremely high [7]. To avoid an exhaustive collision detection test that requires time $O(n^2)$, an spatial-temporal coherence strategy has been implemented. In the case of collision, additional forces are introduced to maintain the system in a legal state.

In this work, the key factors, like data distribution, load balancing and communication overhead, have been considered in order to obtain a high speed-up for the related problem. The application code has been implemented on a cache-coherent shared memory platform (SGI Origin 2000), and may be easily ported to other multiprocessors and multicomputers.

In section 2, a description of the used models and the implementation technique for the implicit integrator are presented. Also, the resolution of the resulting system of algebraic equations, by means of the Conjugate Gradient method, is analyzed. In section 3, the parallel algorithm and the data distribution technique for a scenery is shown. Finally, in section 4, some results and conclusions are presented.

2 Implementation

To create animations, a time-stepping algorithm has been implemented. Every step is mainly performed in three phases: computation of forces, determination of interactions and resolution of the system. The iterative algorithm is shown in figure 1.

The `Update State` procedure computes the new state of the system, which is made up of the position and velocity of each element, calculated from the previous one. `Compute Forces`, `Collision Detection` and `Solve System` stages are described below.

2.1 Forces

Forces and constraints are evaluated on every discrete element in order to compute the equation coefficients for the second Newton's law. For a practical and general application, both spring-mass discretization —of 2D–3D objects, as shown in figure 3— and triangular patches —for the special case of 2D objects like garments, as shown in figure 2— have been included. The particular forces considered are: Visco-Spring forces mapped on the grid for the former model; and Stretch, Shear and Bend forces for the later. In both cases, gravity and air

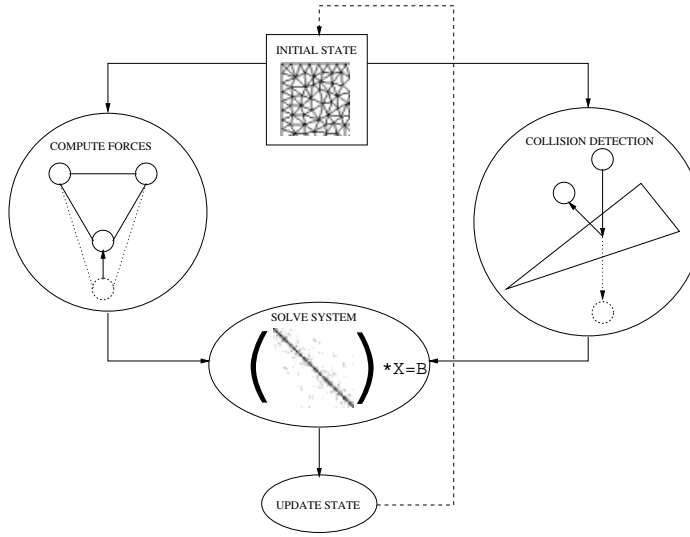


Fig. 1. Simulation Diagram.

drag forces have been also included. The backward Euler method approximates the second Newton's law by the equation (2),

$$\Delta v = \Delta t \cdot M^{-1} \cdot f(x_i + \Delta x, v_i + \Delta v) \tag{2}$$

being $\Delta x = \Delta t(v_i + \Delta v)$. This is a non-linear system of equations which has been time-linearized by one step of the Newton method as follows:

$$f_{i+1} = f(x_{i+1}, v_{i+1}) = f_i + \left|\frac{\partial f}{\partial x}\right|_i \Delta x + \left|\frac{\partial f}{\partial v}\right|_i \Delta v \tag{3}$$

An energy function E_α for every discrete element α is analytically described; the forces acting on particle i are derived from $f_i = -\partial E_\alpha / \partial x_i$ and the arising coefficients from the analytical partial derivations in equation (3) have been coded for its numerical evaluation. All above gives a large system of algebraic linear equations with a sparse matrix A of coefficients.

2.2 Collisions

In the case of cloth simulation, the self-collision detection and the human-cloth collisions may be critical and the computational cost can be extremely high [7]. In order to detect possible interactions and forbidden situations, like colliding surfaces or body penetrations, a hierarchical approach based on the use of bounding-boxes combined with a spatial-temporal coherence strategy have been implemented. In these cases, additional forces are introduced in the system matrix to keep the simulation in a legal state. These forces are included in the system as described above.

In the global scenery, every pair of objects have to be checked to detect if collisions occur. In a hierarchical approach each object is associated with a binary tree of Axis Aligned Bounding Boxes ($AABB$s), in which root node represents a box that enclose the whole object and the leaves enclose only single triangles of the surface. To detect when two objects collide and to determine which pairs of triangles are too close, a simple recursive algorithm can be used. In order to exploit the temporal and spatial coherence a more elaborated algorithm is necessary. In general, when searching collision between two subtrees, two possibilities may occur: the corresponding boxes overlap or not. In the first case the algorithm must go on down the trees until the compared nodes are both leaves, in this case, if the boxes overlap a pair of triangles can be very close or touching them selves, and then it is added to the *possible collision list*. In the other case, the non-overlapped boxes have a minimum distance, and this distance is added to another *non-collision list*. Pair of triangles sharing a vertex will never be an element in any list.

In the following steps, it is not necessary to check every pair of objects and all the subsequent hierarchy. The *possible collision list* is reviewed, if the $AABB$s overlap, the element remains in the list, otherwise, the element is deleted and the calculated distance between the $AABB$s is added to the *non-collision list*. Every element in the *non-collision list* is recomputed using a heuristic, in order to predict when the given $AABB$s do overlap, in a robust way.

Once these lists are filled, every pair of triangles in the *possible collision list* is evaluated to determine if repulsion forces must be added in the matrix A.

The main problem is that the *non-collision list* can grow up to $O(n^2)$ because old collisions are always checked on the following steps. In practice, garments only collide with a small, fixed part of the body, so the lists remain in a manageable size. In any case, if the list grows above a given limit, the collision detection system can be restarted.

2.3 Solver

In the `Solve System` procedure, the unknowns Δv are computed. As stated above, implicit integration methods require the resolution of a large, sparse linear system of equations that must be simultaneously fulfilled. An iterative solver, the preconditioned conjugate gradient (PCG) method, has proven to work well, in practice. This method requires relatively few, and reasonably cheap, iterations to converge [1]. The choice of a good preconditioner can result in a significant reduction of the computational cost in this stage. The Block-Jacobi preconditioner has been chosen for the implementation, because of its good parallel behavior [5]. Due to the nature of the problem, blocks have been formed grouping the physical variables (Δv_x, Δv_y, Δv_z) of the particles, so the block dimension is 3×3. A minimization of the norm $(r^{(i)T} P^{-1} r^{(i)})^{1/2}$, being P the preconditioner matrix and $r^{(i)} = Ax^{(i)} - b$ the residual, is the chosen stopping criterion. Heavier particles are so enforced to be closer to the exact solution.

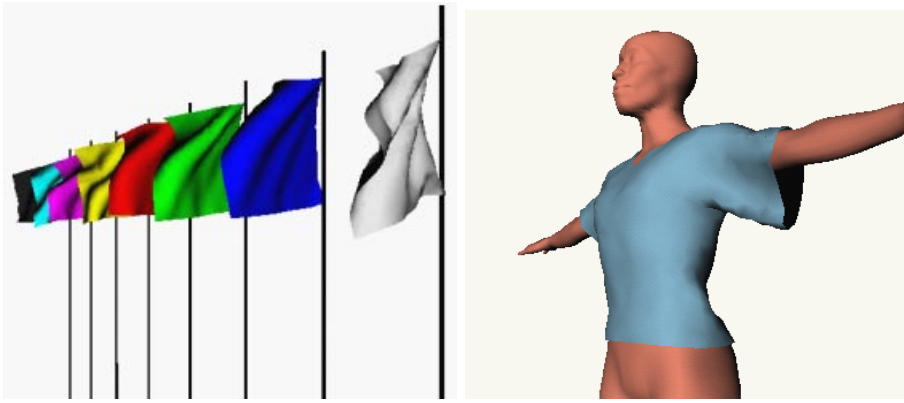


Fig. 2. Flags blowing in the wind and a virtual body wearing a shirt.

3 Parallelization

The parallelization of the model has been performed on a non-uniform memory access (NUMA) multiprocessor architecture. The sequential code (see section 3.1) exhibits irregular access patterns to the data that current parallelizing compilers [4] are insufficiently developed to deal with, leading to non efficient parallel codes. Irregular codes can be parallelized using the inspector-executor paradigm. In run time, the inspector locates non-local data for each processor. Afterwards, an executor must gather non-local data before operate and must scatter the result after it. This strategy introduces a significant overhead, proportional to the number of non-local data accesses. So, a shared memory model and a data parallelism strategy, usual techniques in the *state-of-the-art*, have been used, instead of the run time library. Task parallelism has been also considered for the collision detection stage.

The distribution of the objects in a scenery between the processors is performed using a proportional rule based on the number of elements (particles, triangles,...). The redistribution and reordering of the elements, inside an object among the assigned processors, have been performed using domain decomposition methods. The sparsity pattern of the matrix A is perfectly known, because every non-zero component, with about 12–15 in a row, are the neighbours affecting a given particle for a given tessellation of the object. A Compressed Row Storage (CRS) of the matrix is used in order to minimize memory usage. A striped ordering results in a thin banded diagonal which will produce the parallel distribution with less communication expenses. The Multiple Recursive Distribution (MRD), has higher locality, which will result in a better cache usage [2]. Both have been used in this work with good results, but the choice of the method will depend of the scenery and the computational platform.

3.1 Forces

In the core of the forces evaluation stage, loops like the one presented below[1] are found.

```
for (i=0;i<NumForces;i++) {
   partic0 = List[i][0];
   partic1 = List[i][1];
   force = computeForce(partic0,partic1);
   AccForce[partic0] = AccForce[partic0] + force;
   AccForce[partic1] = AccForce[partic1] - force;
}
```

where `computeForce` does not contain any reference to the array `AccForce[]`. As there are more forces than particles, this is a typical vector reduction operation, and `AccForce[]` is known as *reduction array*. The subscript array `List[][]` depends on the loop index `i`, appearing in both sides in the assignment sentence, through the variables `partic`*i*. In the implicit integration model, both the *rhs* and the system matrix are handled as reduction arrays.

In a parallel environment, two or more processors must synchronize themselves and invalidate their cache copies to actuallize the same memory position (`AccForce`). To overcome this overhead, those particles are replicated in the different processors *id* in a private data structure A_{id}. The loop can now be parallelized, and the final result is the accumulation of the contributions from the processors, $A = \Sigma A_{id}$. A new overhead appears in the accumulation stage, which depends on the number of replicated particles. A good reordering of both `AccForce` and `List` structures reduces the number of replications and minimizes the number of processors where each particle has been hosted.

3.2 Collisions

The lists involved in collisions are distributed among the processors, which compute new contributions to the coefficients of matrix A. As above, such coefficients are replicated in A_{id} to avoid write dependences, and after this step the accumulation into A is done. When collision occurs, the matrix A has a new sparsity pattern because additional coefficients have to be included in unexpected positions. A practical solution is to store them in an additional matrix A^c, also in compressed format.

If lists become longer than a given limit, new lists have to be recomputed from the *AABB* trees. Dealing with hierarchical data structures, it is difficult to use the data parallellism and it is better to keep the sequential code. An additional processor is used to perform this task, without increasing the simulation time. Resulting data are not inmediately required, so the simulation can go on while this task is completed.

[1] Note that this code corresponds with an explicit integration method, but the discussion can be extended to an implicit one.

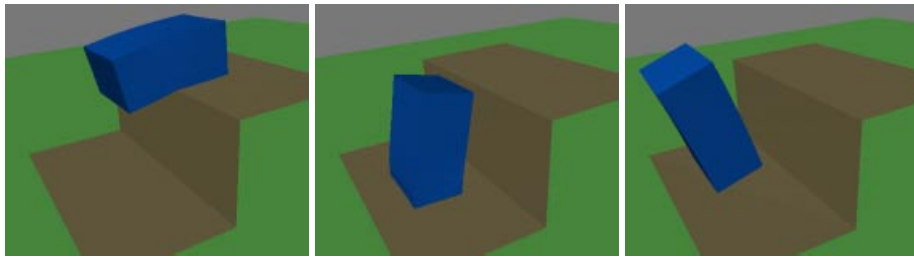


Fig. 3. Some frames of a sponge falling downstairs.

3.3 Conjugate Gradient

The PCG algorithm has been parallelized following a well-known strategy in which the successive parts of the vectors and the properly aligned rows of the matrices are distributed among the processors. This data partition matches with the distribution of the particles in the mesh. Computations inside a processor have been performed using sequential BLAS libraries, which are specially optimized for the underlying hardware.

Using this scheme of PCG [5], very few accesses to remote memories and synchronization points are required along the iterative process. In particular, three global synchronization points, one for every inner product and one for the computation of the convergence criterion, are required. Accesses to remote memories are carried out in these steps and also during the matrix-vector product.

4 Results and Conclusions

Figure 2 shows a human body wearing a shirt and several flags under different windy conditions and figure 3 shows some frames of a simulation of a sponge falling downstairs. In figure 4, the execution time in seconds, of one second of simulated time, and their corresponding efficiency, as a function of the number of processors, for three example models, are shown respectively. These figures correspond to simulations, under windy conditions, of flags of different complexity, with 599, 2602 and 3520 particles respectively. Each curve in a graph corresponds to the original unsorted data, MRD and striped sort distributions. These results has been obtained using an SGI Origin2000 computer with R10000-250Mhz processors. A real time simulation is obtained for the former model, with six processors using striped ordering. The efficiency of the third model shows a superlinear speed-up, which is a consequence of the increment of the ratio computation/communication. It can be observed that striped distribution is clearly faster than MRD for more than two processors, due to the minimization of the accumulation stage overhead.

The computational load of this problem is heavy enough to obtain good efficiencies, even for the simplest models. For larger models, the speed-up grows and gets linear, when problems of typical complexity are dealt with. A proportional

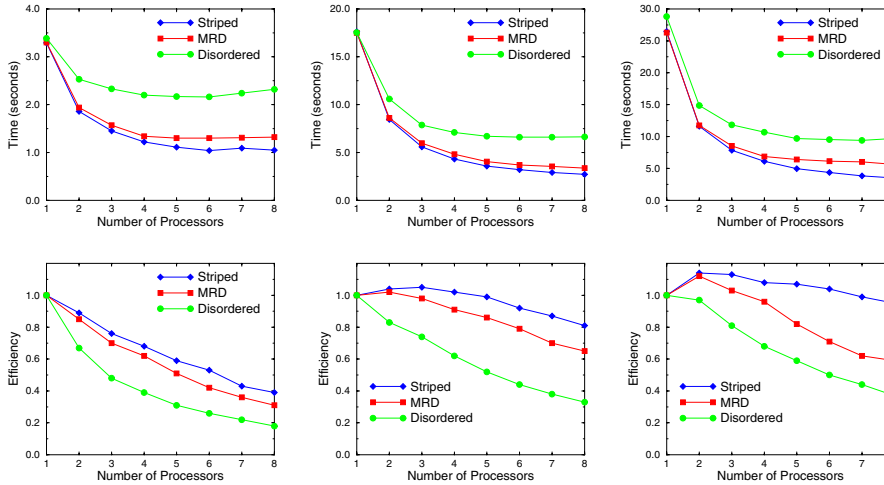


Fig. 4. Execution Time, Speed-up and Efficiency graph for 599, 2602 and 3520 particles.

increment of the number of processors and the size of the problem keeps the efficiency in an almost constant value. This property (isoefficiency, [3]) ensures the validity of the presented model for large simulations.

The use of more recent computers and a higher number of processors for models with more particles/triangles will allow real time simulations. The scenery complexity, considering interaction between several objects, will be improved as the speed of the microprocessors increases.

References

1. Baraff, D., Witkin A.: Large Steps in Cloth Simulation. In Michael Cohen, editor, Computer Graphics (SIGGRAPH 98 Conference Proceeding), pages 43–54. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.
2. Romero, L.F., Zapata E.L.: Data Distribution for Sparse Matrix Vector Multiplication. Parallel Computing Vol. 21, pp. 583–605, 1995.
3. Gupta, A., Kumar, V., Sameh, A.: Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers. IEEE Trans. on Parallel and Distr. Systems, Vol. 6, No. 5, pp. 455–469, 1995.
4. Silicon Graphics Inc. MIPSpro Auto-Parallelizing Option Programmer's Guide.
5. Dongarra, J., Duff, I.S., Sorensen, D.C., Van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers Software, Environments and Tools series. SIAM, 1998.
6. Volino, P., Courchesne, M., Thalmann, N.: Versatile and efficient techniques for simulating cloth and other deformable objects. Computer Graphics, 29 (Annual Conference Series):137–144, 1995.

7. Volino, P., Thalmann, N.: Collision and Self-collision detection: Efficient and Robust Solutions for Highly Deformable Objects. Computer Animation and Simulation'95: 55–65. Eurographics Springer-Verlag, 1995.

The Input, Preparation, and Distribution of Data for Parallel GIS Operations

Gordon J. Darling[1], Terence M. Sloan[1], and Connor Mulholland[1]

EPCC, University of Edinburgh, EH9 3JZ, UK,
http://www.epcc.ed.ac.uk/

Abstract. Geographical Information Systems (GIS) manipulate spatial data from a variety of data formats. The widely adopted vector-topological format retains the topological relationships between geographical features and is typically used in a range of geographical data analyses. There are a number of characteristics of the format, however, that cause difficulties in the input, manipulation and processing of the data. This paper reports on the performance of a prototype parallel data partitioning algorithm for the input of vector-topological data to parallel processes.

1 Introduction

The continued rapid growth in the availability of digital and cartographic data and satellite images is creating a demand for intensive computing to integrate and process large datasets. Of particular interest to organisations working in the GIS field is their ability to quickly and efficiently process these large datasets by maximising the performance of their existing hardware. Commonly, datasets can consist of around 100,000 polygons and in some cases, for example zip code data, considerably more (1.5 million polygons). The processing of these present large-scale computational difficulties that are of increasing importance to the GIS community [1] - eg. the rapid production of detailed maps for disaster management or simulation. Typically, however, organisations working within the GIS field do not have access to supercomputing facilities and their available hardware consists of networks of PCs or workstations. Beowulf systems [2] and the use of networks of workstations are, therefore, fields of research that are highly pertinent to the needs of the GIS community.

This paper reports on the performance of a prototype algorithm based on designs described in [3] and its success in meeting challenges posed by the processing of vector-topological data. Although the designs are applicable to a variety of platforms, the results reported here have been generated from their implementation on a network of workstations at EPCC. The paper briefly reviews the structure of vector-topological data and describes the importance of the initial processing of this data. A description of the implemented algorithm and of issues affecting its design and implementation are presented. Some initial results are then discussed.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 500–505, 2000.
© Springer-Verlag Berlin Heidelberg 2000

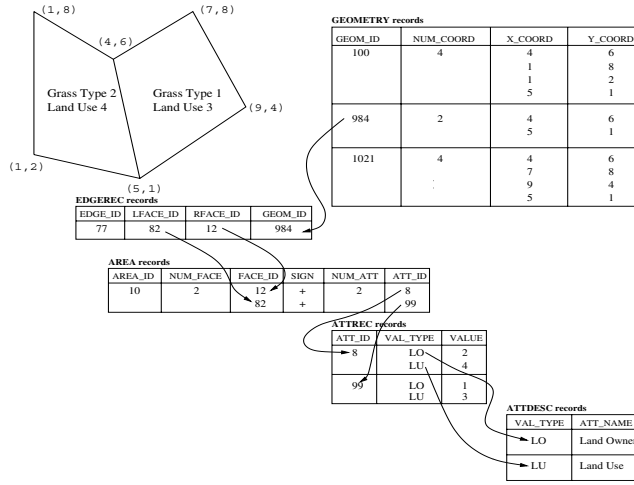


Fig. 1. An example of two adjoining polygons and the use of unique IDs to relate a subset of records describing the topology. These comply with the NTF standard representation of vector topology [4].

2 Vector-Topological Data

A vector-topological representation uses geometrical objects (eg. points, lines and polygons) to represent each feature. Of importance to the processing of this data is the accompanying separation of spatial information and other attributes. Figure 1 illustrates how unique identifiers are employed to link the data in a relational way with a resulting, complex data structure emerging. Thus, the data model is not flat but depth is provided by multiple records, interrelated by the identifiers. Operations on this type of data model follow several links between records to gather the data which, in turn, often require to be sorted prior to further processing. Consequently, the handling of such data is non-trivial. The processing of vector-topological data is further complicated by the variable-lengths of records - eg. in Figure 1, the records `GEOM_ID 100` and `GEOM_ID 984` are of different lengths. In general, the variable length of records has implications on the input and output of data, its processing and its communication across parallel processes. These are discussed in some detail in [3].

3 The Parallel Data Partitioning Algorithm

The relational format of vector-topological data requires the extraction of information by following links between multiple records. The reading, sorting, merging and distribution of data is therefore of great importance to the efficiency of parallel GIS operations [3, 5].

To extract the spatial information from unsorted input records and to spatially decompose and distribute the data for efficient processing, the algorithm proceeds through two phases: **Data Preparation**, where the spatial sorting and merging of records takes place, and **Data Distribution**, known as the *GAD phase*, where the decomposition of the data is performed according to spatial attributes. See [3, 6, 7] for comprehensive details of the operation.

The data preparation phase comprises the *Sort* and *Join* phases. In the Sort phase unsorted records are read by multiple processes. A single *Source process* coordinates access to the input file(s) and sends a message enabling *Worker processes* to extract the appropriate attribute and spatial information. Each Worker reads and processes data and places them in an internal parcel according to their record origins. As a by-product, Workers generate lists that describe the spatial distribution of the extracted and sorted data. At the end of the Sort phase, the Source produces a final, merged list that is used in the decomposition and distribution of data to processes. The Join phase is responsible for merging the various record files produced within the Sort phase.

Within the data distribution (GAD) phase, the list produced by the Source process is used to determine a decomposition of the dataset into regions of workload reflecting the processing capacities of Worker processes. Feature boundaries (generated by Sort) and their associated attribute values (generated by Join) are distributed to processes. Regions are distributed across the available parallel processes, one per process.

4 Implementation and Performance

The prototype Vector Input algorithm was implemented in ANSI C with the MPICH 1.1.1 version [8] of the MPI standard [9] and run on a network of Sun Ultra 5 workstations. The results reflect the improvements in algorithm performance and functionality from those reported by Sloan *et al* [7]. In particular, the GAD phase is now fully operational and initial performance figures have been gathered on a network of workstations and on a shared-memory platform [6]

Table 1 indicates the averaged times for the processing of two replica 2.44 Mb vector-topological datasets, measured from initialisation of the Sort phase through to the completion of the GAD distribution phase. The input of multiple datasets is a requisite stage in computationally intensive GIS operations such as *Polygon Overlay* [10]. Currently, the algorithm requires a minimum of eight processes to be run [6, 7]. The processes are distributed across the available processors. In our example, the input datasets resided on a SCSI Ultra disk on a Sun E450 with a 10 Mbit/sec network connection. The most notable features of the data presented in Table 1 are the domination of the Sort phase to the overall processing time and the rapid reduction in the elapsed time to Sort the data when more than a single processor is utilised. The intensive I/O demands of the Sort phase clearly indicate the necessity for a parallel I/O utility to be developed. Currently, a single Source process is required to handle all read access to the input datafiles for all the other processes and to communicate the appropriate data

to workers. Initial investigations of utilising MPI-IO from the MPI-2 standard [11] have, however, proved inconclusive [12]. Secondly, the extremely large reduction in processing times with the introduction of a second processor indicate that issues of process swapping and the distribution of processes across available processors require to be further assessed. However, the results displayed in Table 1 validate the potential of the parallel algorithm in enhancing the throughput and processing of vector-topological data. In the current implementation, however, it is useful to examine the reliance of the algorithm on features such as disk performance, data location and on process-to-processor assignment. In addition, there is scope for further improvements through tuning and performance analysis. This work and a full discussion of the results presented in Table 1 are available at `http://www.epcc.ed.ac.uk/research/ParaGIS/EPSRC/index.html`.

Table 1. Elapsed times in seconds for the Sort, Join and GAD phases and the overall processing time, applied to the processing of two replicates of the New Boone dataset [13] on a network of Sun Ultra 5 workstations: (a) no local disk (b) no local disk where the Source process has a dedicated processor. (c) Local disk to the Source process utilised, where the Source process has a dedicated processor.

No. Utilised Processors	1	2	3	4	5	6	7	8
Sort								
(a)	3866	390	280	218	217	217	227	159
(b)	3866	386	152	155	155	154	159	159
(c)	3857	161	142	146	145	145	151	151
Join								
(a)	39	29	28	24	24	24	23	19
(b)	39	28	27	24	22	22	22	19
(c)	34	31	28	23	22	23	23	21
GAD								
(a)	32	18	16	12	11	11	10	6
(b)	32	21	14	13	12	12	11	6
(b)	31	24	13	12	11	11	11	8
Overall								
(a)	3937	437	324	254	252	252	260	184
(b)	3937	435	193	192	189	188	192	184
(c)	3922	215	183	181	178	179	185	180

5 Conclusions and Future Work

The successful processing of vector-topological datasets has been performed, in parallel, from input through to their distribution. The prototype algorithm has been implemented successfully and has been shown to handle the complex data structures arising within a typical vector-topological dataset. The processing

within the Sort phase dominates processing times and areas where improvements to the algorithm's performance may be made have been identified. Particularly, within the Sort phase, there is scope for tuning—eg. buffer sizes; server/worker ratios etc.—and a parallel I/O facility would greatly benefit the input of the data. Nonetheless, our prototype parallel implementation of the input of vector-topological data provides evidence of the applicability of a parallel approach to a crucial first step in many GIS operations.

6 Acknowledgements

The authors thank Mike Mineter, Steve Dowers and Bruce Gittings of The Department of Geography, University of Edinburgh for their contributions and acknowledge the support of EPSRC in funding this work.

References

[1] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *ACM Symposium on Computational Geometry*, 1998.

[2] T. Sterling and D. Savarese. A coming agenda for beowulf-class computing. In Amestoy K. et al, editor, *Lecture Notes in Computer Science: Euro-Par'99 Parallel Processing. Proceedings of the 5th International Euro-Par Conference,* `Toulouse, France`, volume 1685, pages 78 – 88, 1999.

[3] T M. Sloan and S. Dowers. Parallel Vector Data Input. In R. G. Healey, S. Dowers, B. M. Gittings, and M. J. Mineter, editors, *Parallel Processing Algorithms for GIS*, chapter 8, pages 151 – 178. Taylor and Francis, 1998.

[4] British Standards Institution. *Electronic transfer of geographic information (NTF) Part 1. Specifi cation for NTF structures*, BS 7567 : Part 1 : 1992 edition, 1992.

[5] M. J. Mineter and S. Dowers. Parallel processing of geographical applications: A layered approach. *J. Geograph Syst*, 1(1):61 – 74, 1999.

[6] G. J. Darling, C. Mulholland, T. M. Sloan, M. J. Mineter, S. Dowers, and B. M. Gitting. Parallel input of vector topological data to gis operations. Submitted to *Concurrency: Practice and Experience*, 2000.

[7] T. M. Sloan, M. J. Mineter, S. Dowers, C. Mulholland, G. J. Darling, and B. M. Gittings. Partitioning of vector-topological data for parallel gis operations: Assessment and performance analysis. In Amestoy K. et al, editor, *Lecture Notes in Computer Science: Euro-Par'99 Parallel Processing. Proceedings of the 5th International Euro-Par Conference,* `Toulouse, France`, volume 1685, pages 691 – 694, 1999.

[8] W. Gropp and E. Lusk. *User's guide for MPICH, a Portable Implementation of MPI.* Mathematics and Computer Science Divison, Argonne National Laboratory, University of Chicago, USA, anl/mcs-tm-anl-96/6 edition, 1998.

[9] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee,, U. S.. A. *MPI; A Message-Passing Interface Standard*, 1.1 edition, June 1995.

[10] T J. Harding, R G. Healey, S. Hopkins, and S. Dowers. Parallel Vector Polygon Overlay. In R. G. Healey, S. Dowers, B. M. Gittings, and M. J. Mineter, editors, *Parallel Processing Algorithms for GIS*, chapter 13, pages 265 – 310. Taylor and Francis, 1998.

[11] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee,, U. S.. A. *MPI-2: Extensions to Message-Passing Interface*, 1.1 edition, July 1997.
[12] E. Moita. Optimisation of parallel vector-topological data input for Geographical Information Systems using MPI-IO. Technical Report EPCC-SS98-9, EPCC, 1998.
[13] US Bureau of Census. Extract from the prototype TIGER/Line File for Boone County, Missouri, 1988.

Study of the Load Balancing in the Parallel Training for Automatic Speech Recognition*

El Mostafa Daoudi[1], Pierre Manneback[2], Abdelouafi Meziane[1], and Yahya Ould Mohamed El Hadj[1]

[1] Université Mohammed Ier, Faculté des Sciences, LaRI, 60 000 Oujda, Morocco
{mdaoudi,meziane,h.yahya}@sciences.univ-oujda.ac.ma
[2] Faculté Polytechnique de Mons, Rue de Houdain, 9, 7000 Mons, Belgium
Pierre.Manneback@fpms.ac.be

Abstract. In this paper we propose a parallelization technique of the training phase for the automatic speech recognition using the Hidden Markov Models (HMMs), which improves the load balancing in the previous proposed parallel implementations [1]. This technique is based on an efficient distribution of the vocabulary on processors taking into account, not only the size of the vocabulary, but also the length of each word. In this manner the idle time will be reduced. The experimental results show that good performances can be obtained with this distribution.
Key words: automatic speech recognition, Markovian modeling, parallel processing, load balancing.

1 Introduction

Many approaches have been proposed to solve the automatic speech recognition problem (ASR). At the present time, the most efficient and the most used systems of recognition are based on the Hidden Markov Models (HMMs) [6]. However the algorithms relating to these models are very expensive in terms of computation time and memory space. To our knowledge, only a small number of works related to the parallelization of ASR systems have been proposed in the literature [7,4,8]. We note that it is very important to be able to make fast training since the performance of an ASR system depends on the high quality of this phase. In this work, we propose a parallel implementation on a distributed memory machine of the training phase using the widespread framework, which explicitly builds the global Markovian network by calling upon linguistic knowledges structured at various levels (acoustic level, phonetic level ...etc) [6]. In this implementation two distribution strategies are used, based on an uniform distribution of the vocabulary on processors. The first distribution assigns randomly words to processors, the latter does the word to processor assignment taking into account their training costs.

* This work is supported by the Keep-In-Touch Project 972644 "DAPPI", INCO-DC Program, DG III, Commission of the European Communities.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 506–510, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 The Training

The parameters of the model are estimated from a training set composed by various pronunciations of each word of the vocabulary. These parameters are:

- The probability of transitions between the states $(q_i)_{1\leq i\leq N}$ of the model: $A = (a_{ij})_{1\leq i,j\leq N}$, where N is the number of states of this model.
- The probability distributions governing the emission of acoustic observations from states of model: $B = \Big(b_i(.)\Big)_{1\leq i\leq N}$
- The initial probability distribution: $\Pi = (\pi_i)_{1\leq i\leq N}$.

An HMM model is represented by $\lambda = (\Pi, A, B)$. The training of these parameters is done by an iterative re-estimation procedure using a best path research algorithm (Baum-Welch or Viterbi). We have chosen the Viterbi algorithm [5], which is the most common and has the lower computation time. This algorithm researches the path which has the most plausibly generated a sequence of observations $Y_T = y_1, \cdots, y_T$ in the model λ. For each state of the model and for each observation y_t, we consider the recurring formula:
$\delta_t(q_j) = \max\limits_{i\in Pred(j)} \Big(\delta_{t-1}(q_i) \times a_{ij}\Big) \times b_j(y_t)$, where $Pred(j)$ indicates the predecessors set of the state q_j.

This expression represents the highest probabilities to send the sequence of observations, $y_1, ..., y_t$, on the set of partial paths of length t and of final state q_j.

In order to obtain the states of the optimal path, we also use another variable, ψ_t, in which we store, at every time t, the state which determines the value of $\delta_t(q_j)$.

The value of δ_T, at a final state of model, determines the emission probability of Y_T conjointly to the optimal path, as well as the final state of this path. The other states of the optimal path are obtained by backtracking using the ψ_t variable.

3 Complexity of the Training

According to the hierarchical construction of the network, each phonetic unit ϕ_τ is represented by a Markovian model, which is chosen in left-to-right type. Each state q_j of this model has, at most, j predecessors in the same model and np predecessors in the models of the phonetic units directly attached to ϕ_τ. The number of the states of ϕ_τ will be denoted by N^{ϕ_τ}.
First of all, we determine the number of floating operations (*flops*) performed in a phonetic unit ϕ_τ, noted by $T_{cal}^{\phi_\tau}$, relating to the Viterbi algorithm. We compute the $\delta_t(q_j)$ variable, for all states $(q_j)_{1\leq j\leq N^{\phi_\tau}}$ and for all observations $(y_t)_{1\leq t\leq T}$.

The computation of the acoustic law $b_j(y_t)$ requires a fixed number of *flops*, which will be noted by C_1^{te}. The quantity $\max\limits_{i\in Pred(j)} \big(\delta_{t-1}(q_i) \times a_{ij}\big)$ requires:

$Card\big(Pred(j)\big)$ *flops* and $Card\big(Pred(j)\big) - 1$ comparisons.

For fixed t and j, the number of *flops* to compute $\delta_t(q_j)$ is $2\,Card\big(Pred(j)\big)+C_1^{te}$

Hence $T_{cal}^{\phi_\tau} \simeq \sum_{t=1}^{T}\sum_{j=1}^{N^{\phi_\tau}}\left(2\times Card\Big(Pred(j)\Big)+C_1^{te}\right)$

Developing this expression, we obtain: $T_{cal}^{\phi_\tau} \simeq \Big(2np+N^{\phi_\tau}(N^{\phi_\tau}+C_1^{te}+1)\Big)T$

Let ϵ_i denote the number of phonetic units of the model of a word number i, including the starting an ending silences wrapping the word. Using $T_{cal}^{\phi_\tau}$, we can estimate the number of *flops* necessary for training on this word:

$\sum_{\tau=1}^{\epsilon_i}\left(2np+N^{\phi_\tau}\left(N^{\phi_\tau}+C_1^{te}+1\right)\right)T_i^j \simeq \epsilon_i\left(2np+\overline{N}_i^{\phi_\tau}\left(\overline{N}_i^{\phi_\tau}+C_1^{te}+1\right)\right)T_i^j$,

where $\overline{N}_i^{\phi_\tau}=\frac{\sum_{\tau=1}^{\epsilon_i}N^{\phi_\tau}}{\epsilon_i}$ is the average number of states per phonetic unit relating to the word i and T_i^j the number of observations of this pronunciation. We remark that $N_i = \epsilon_i\overline{N}_i$ represents the number of states of the sub-network of word i. Thus the training cost $C_{tr}(i)$ of a word i pronounced n_i times is:

$$C_{tr}(i)\simeq\sum_{j=1}^{n_i}\left(2np\epsilon_i+N_i\left(\overline{N}_i^{\phi_\tau}+C_1^{te}+1\right)\right)T_i^j=\left(2np\epsilon_i+N_i\left(\overline{N}_i^{\phi_\tau}+C_1^{te}+1\right)\right)T_i$$

where $T_i=\sum_{j=1}^{n_i}T_i^j$ is the number of observations for all pronunciations of this word.

4 Parallelization

We consider a distributed memory architecture composed of p processors numbered by $(P_i)_{0\le i\le p-1}$. In previous works, we have proposed a parallelization of the HMM models [1] and the centisecond TLHMM models [2,3], based on the duplication of the network. In the course of these works, we have remarked that the manner to assign the words on processors plays a very significant role in the performances of proposed algorithms. This comes essentially from the difference in length in the learned words and from the way they are distributed on processors. Indeed, it is possible that a processor deals only with the shortest words, while another deals with the longest ones. In this study, we propose an improvement of [1] by using a distribution based on the training cost of each word.

Parallelization strategy: we use a technique of duplication of the network, which consists in assigning to each processor a copy of the global Markovian network. The set of m words, representing the vocabulary, is uniformly distributed on processors. This distribution can be randomly performed by affecting $\frac{m}{p}$ words to each processor independently to their lengths. However, a better strategy consists in using training complexities in order to obtain a better load balancing. The training cost $C_{tr}(i)$ (paragraph 3) is calculated for each word and is stored in a decreasing way into a vector C. The vocabulary is next distributed by using a cyclic permutation on processors. For example, a cyclic permutation on 3 processors is given by:

$$\begin{array}{llll} C_0\ C_1\ C_2 & C_3\ C_4\ C_5 & C_6\ C_7\ C_8 & \cdots \\ P_0\ P_1\ P_2 & P_2\ P_0\ P_1 & P_1\ P_2\ P_0 & \cdots \end{array}$$

Each processor carries out the training of the local corpus composed of $\frac{m}{p}$ words, where each one is pronounced n_i times. After simultaneous local training, an exchange between all processors is performed to re-actualize the global network. During this phase of communication, which is of *all-to-all* type, a processor can anticipate the re-estimation of the probabilities of its own models. The re-estimation of the acoustic laws as well as the remaining of the global network are done after this communication.

Complexity: since the global network is duplicated on all processors, the iteration computation time for the re-estimation is identical to the corresponding sequential processing time. If we assume that the training cost is independent of the word, then the iteration computation time to learn the local words is equal to the computation time to learn sequentially all words divided by p.

For the communication, we determine an upper bound of the volume of exchanged data by using the maximum number of laws and transitions on an optimal path. If a pronunciation of the word i is composed by T_i^j observations, then the optimal path associated with this pronunciation contains, at most, $min(T_i^j, 2N_i - 1)$ different transitions. N_i is the number of states of the sub-network of this word. A transition is characterized by some parameters and by an acoustic law. The acoustic laws, that we use, are multi-gaussian of mean vectors of length μ and diagonal covariance matrices of size μ. It follows that the volume of exchanged data generated by the local training is, at most, equal to $\sum_{i=1}^{\frac{m}{p}} \sum_{j=1}^{n_i} \left(\left(2\mu + C^{te}\right) \times min(T_i^j, 2N_i - 1) \right)$, where Cte denotes the number of parameters which characterize a transition.

5 Experimentations

The experimentations have been carried on on a vocabulary of 40 words; each word is pronounced by 6 speakers. These data are extracted from the data base BDSONS of French sounds. The parallel programs have been developed under the PVM environment on a distributed memory Computer Telmat TN310 composed of 32 T9000 transputers. In table 1, we report the average computation time, by iterattion, of the training relating to both the random and the proposed distributions with different number of processors ($p = 10$ and $p = 20$). These times are given for the fastest and the slowest processors. This table shows a significative load imbalance between processors for the random distribution, but this is largely improved by the proposed distribution. In table 2, we give the average execution time of one iteration of the training relating to the random and the proposed distributions, obtained with different number of processors. It shows the impact of the distribution on the algorithm performances. Experimental results indicate that good performances are obtained with the latter distribution, which corroborate the theoretical analysis.

distribution	random				studied			
nbre of processors	$p = 10$		$p = 20$		$p = 10$		$p = 20$	
processors	slowest	fastest	slowest	fastest	slowest	fastest	slowest	fastest
time / s	15.32	5.98	7.92	2.86	11.81	9.98	6.89	4.30

Table 1. *Computation times, by iteration, for the slowest and the fastest processors*

training	sequential	parallel					
nbre of processors	1	2	4	5	8	10	20
random distribution	83.374	73.210	38.054	30.959	20.144	16.498	9.450
proposed distribution	83.374	59.036	29.146	23.278	15.390	13.000	8.457

Table 2. *Execution time, by iteration, of the training*

6 Conclusion

In this work we have proposed a parallel implementation for the training phase of the automatic speech recognition, based on the duplication of the network on all processors. In this implementation two distribution strategies, based upon an uniform distribution of the vocabulary on processors, are exploited. The first one (random distribution) consists in assigning words to processors independently to their lengths, whereas the second one consists in assigning words to processors taking into account their training cost. Experimental results show that good performances can be obtained with the second strategy, the load imbalance being largely improved. We intend to exploit this result in the case of network distribution [3].

References

1. E.M. Daoudi, A. Meziane, Y.O. Mohamed El Hadj, "Study of Parallelization of the Training for Automatic Speech Recognition", HPCN Europe 2000, LNCS 1823, pages 576-579.
2. E.M. Daoudi, A. Meziane, Y.O. Mohamed El Hadj, "Parallel training for the automatic speech recognition using the centisecond TLHMM model", ACIDCA'2000, pages 142-147 (Vision and Pattern Recognition), Tunisia 2000.
3. E.M. Daoudi, A. Meziane, Y.O. Mohamed El Hadj, "Parallel HMM model for automatic speech recognition", RR-LaRI, Oujda, 1999, submitted for publication.
4. M. Fleury, A. C. Downton, A. F. Clark, "Parallel Structure in an Integrated Speech-Recognition Network", EuroPar'99, LNCS 1685, pages 995-1004, 1999.
5. D. R. Forney, "The Viterbi Algorithm", Proc IEEE, Vol 61, $n^\circ 3$, Mai 1973.
6. A. Meziane, "Introduction de la durée des sons dans un modèle de Markov caché au niveau supra segmental", Thèse de doctorat d'état, Université Oujda, Avril 1997.
7. H. Noda, M. N. Shirazi, "A MRF-based parallel processing algorithm for speech recognition using linear predictive HMM", ICASSP'94, pages I-597 - I-600, 1994.
8. S. Phillips, A. Rogers, "Parallel Speech Recognition", EUROSPEECH'97, 1997.

Pfortran and Co-Array Fortran as Tools for Parallelization of a Large-Scale Scientific Application

Piotr Bała[1,2] and Terry W. Clark[2]

[1] Interdisciplinary Centre for Mathematical and Computational Modelling, Warsaw University, Pawińskiego 5a, 02-106 Warsaw, Poland,
bala@icm.edu.pl
[2] Institute of Physics, N. Copernicus University, Grudziądzka 5/7, 87-100 Toruń, Poland,
[3] Department of Computer Science
The University of Chicago and Computation Institute
1100 E. 58th Street, Chicago, IL 60637, USA
twclark@cs.uchicago.edu

Abstract. Parallelization of scientific applications remains a nontrivial task typically requiring some programmer assistance. Key considerations for candidate parallel programming paradigms are portability, efficiency, and intuitive use, at times mutually exclusive features. In this study two similar parallelization tools, Pfortran and Cray's Co-Array Fortran, are discussed in the parallelization of Quantum Dynamics, a complex scientific application.

1 Introduction

Today's parallel computers routinely provide computational resources permitting simulations based on complex scientific models such as the models for describing the dynamics of quantum systems [1]. This area has experienced heightened activity with the recent progress in experimental physics, especially ultrafast optical spectroscopy and quantum electronics. Because analytical tools available for the description of quantum dynamical systems are limited, computational models must be used. Quantum dynamics simulations are usually based on numerical propagation of a quantum wavefunction obeying the time-dependent Schroedinger equation. This task can be easily performed for one-dimensional systems, but in most cases, the size of the system is limited by the available computational resources, i.e., computer memory and speed. However, this obstacle can be overcome with parallel computers.

For the task of parallelization, well-established libraries such as MPI [2], PVM [3,4] and *shmem* [5] can be used. An important advantage is availability of implementations of these on a wide range of hardware platforms. There remains, however, significant barriers to parallelizing complex scientific applications because complicated applications resist automatic parallelization, while low-level

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 511–518, 2000.
© Springer-Verlag Berlin Heidelberg 2000

methods of parallelization lead the applicationist into complex and fragile methods obscuring the algorithmic intent behind the code.

To fill the gap between high-level approaches such as HPF and low-level approaches such as MPI, intermediate-paradigms have been developed which address important issues of efficiency, concise notation, and access to low-level details. In this paper we consider two of these: Cray's Co-Array Fortran [6,7] and Pfortran [8,9]. The aim of this paper is to compare these intermediate-level tools and their efficacy for parallelizing large-scale scientific applications starting from both specially-defined test cases and production code.

2 Quantum Dynamics Algorithm

We parallelized the Quantum Dynamics (QD) code described in [10]. In QD, the dynamics of the quantum particle is given by the time-dependent Schroedinger equation modeled with a discrete representation of the wavefunction on a uniform Cartesian grid. A Chebychev polynomial method is used for the propagation of the wavefunction [1,11]. Simulations consist of the evolution of the wavefunction, which is evaluated at each timestep during the simulations.

The propagation of the quantum-particle wavefunction requires at each time step several evaluations of the Hamiltonian acting on the wavefunction. In practical calculations, both wavefunction and potential are represented on the discrete grid and all variables are calculated numerically on the grid.

The evaluation of the potential energy part of the Hamiltonian is a relatively lightweight computation consisting of the multiplication of a wavefunction by the value of the potential. The evaluation of the kinetic part $\frac{1}{2m}\Delta_x\Psi(x)$ is performed using a Fast Fourier Transform (FFT) [12]. We used a three-dimensional parallel Fast Fourier Transform, PCFFT3D, from the Cray T3E Scientific libraries for this. PCFFT3D requires a slicewise distribution of the transformed matrix over processes.

3 Parallelization Tools

Co-Array Fortran and Pfortran were used in the parallelization of the QD code. Both languages fall into the SPMD program model with all processes executing the same program. Parallelism is exploited by explicit partitions of data and control flow, or some combination of the two. Operations involving arrays can be easily performed in parallel with the programmer distributing arrays and iterations across the processors, allocating only the memory required for the local part of the array if necessary. Ordinary variables are replicated, with the scalar parts of the code performed independently and redundantly on each processor.

3.1 Pfortran

Pfortran extends Fortran with several operators designed for intuitive use and concise specification of off-process data access. *PC* is the C counterpart to Pfortran; both are implementations of the Planguages. The crux of Pfortran and

PC are *fusions*, a variant of the guarded-memory model [13]. *Fusion objects* are distributed variables formed syntactically with the Planguage operators @ and {}.

In a sequential program the assignment statement specifies a move of a the value at the memory location represented by j to the memory location represented by i. The Planguages allow the same type of assignment, however, the memory need not be local, as in the following example in a two-process system

$$i@0 = j@1$$

with the intention to move the value at the memory location represented by j *at process 1* to the memory location represented by i *at process 0*.

The other Pfortran fusion operator consists of a pair of curly braces with a leading function, $f\{\}$. This operator lets one represent in one fell swoop the common case of a reduction operation where the function f is applied to data across all processes. For example, suppose one wanted to find the sum of an array distributed across *nProc* processes, with one element per process. One could write

$$sum = +\{a\}$$

where a is a scalar at each process, but can be viewed logically as an array across *nProc* processes. With @ and {}, a variety of operations involving off-process data can be concisely formulated.

In the Planguage model, processes can interact *only* through the same statement, with such statements containing one or more fusion objects. Synchronization is implied along with the @ and {}. A consumer will obtain off-process data as it is defined at the point in the program where the access is performed, i.e., at the @ and {}. If there is uneven progress by the consumer and producer, the implementation could either buffer the data, or stall the producer.

Programmers have access to the local process identifier called *myProc*. With *myProc*, the programmer distributes data and computational workload. The Planguage translators transform user-supplied expressions containing fusion objects into algorithms with generic calls to a system-dependent library using, for example, MPI [2], PVM [3], TCGMSG [14] and the Intel message-passing interface. To port a Pfortran code from one machine to another, one simply recompiles it with Pfortran, next compiling the output FORTRAN77 with the native Fortran compiler. This allows for mixing Pfotran code code with traditional Fortran77 subroutines and functions which allows for easy parallelization of large parts of the code.

Pfortran currently targets on the Cray T3E/T3D, IBM SP2, SGI workstations, SUN multiprocessor computers, as well as on clusters of workstations.

3.2 Co-Array Fortran

Cray Co-Array Fortran is the other parallelization paradigm considered in this study [6,7]. Co-Array Fortran introduces an additional array dimension for arrays distributed across processors. For example, the Pfortran statement

$$a(i)@0 = b(i)@1$$

is equivalent with the Co-Array Fortran statement

$$a(i)[0] = b(i)[1].$$

We note that all Pfortran data fusion statements can be written with co-arrays, however, the converse is not true. Also, variables used in Co-Array Fortran statements must be explicitly declared as co-arrays. While the co-array and fusion constructs support the same type of data communication algorithm, Co-Array Fortran generally requires more changes in the legacy code than does Pfortran; however, Co-Array Fortran provides structured distribution of user-defined arrays. Co-Array Fortran does not supply intrinsic reduction-operation syntax. These algorithms must be coded by the user using point-to-point exchanges.

While Co-Array Fortran and Planguage models are similar, they have fundamental differences, namely, with Co-Array Fortran:

1. [ ] does not imply synchronization; the programmer must insert synchronization explicitly to avoid race conditions and data consistency.
2. Inter-process communication with co-arrays *can* occur between separate statements.
3. Co-array variables must be explicitly defined in the code.

The communication underlying Co-Array Fortran is realized through the Cray's *shmem* library, providing high communication efficiency. Cray's parallel extensions to Fortran are available only on selected CRAY architectures limiting the portability of Co-Array Fortran applications.

4 Code Parallelization

The QD application consists of several different types of calculations which we addressed independently for purposes of parallelization. Most parallelization is concerned with the distribution of data and calculations for the wavefunction propagation and potential function evaluation. The most time-consuming part of the code computes the potential and propagates the wavefunction on a three-dimensional spatial grid. The evaluations of the potential and wavefunction propagation at each grid point require *only local* information, so all variables evaluated on the distributed grid, such as the potential $V(x, y, z)$ and the wavefunction $\Psi(x, y, z, t)$, are evaluated in parallel. This step does not involve any communication once arrays are distributed.

The $N_x \times N_y \times N_z$ grid on which the potential and wavefunction are defined are mapped onto a $np_x \times np_y \times np_z$ logical processor array. The mapping is such that processes hold equally sized parts of the grid within a modulo factor. In practice, three-dimensional arrays are linearized to one-dimensional arrays of length $N_{all} = N_x \times N_y \times N_z$. The workload is already balanced using the uniformly distributed grid since the processes perform identical operations at each grid point.

Evaluation of different mean values characterizing quantum particles, such as the energy, position, momentum and norm of the wavefunction requires various reduction operations. These properties are computed only once per timestep, but, because of the communication involved, this step can significantly impact the code performance. In Pfortran and Co-Array implementations, the partial sums are performed at each processor with a subsequent summation of the partial sums across all processors. For this purposes the Pfortran global-sum fusion operator is used with the reduction algorithms generated by the translator; Cray's Co-Array Fortran requires honing an algorithm consisting of point-to-point exchanges.

Parallel I/O consists of inputting and outputting the wavefunction and potential-energy arrays. These quantities are stored in a file in some order required for other applications, so that processes must output their data in appropriate order. This is done with processes accessing files directly, or by individually routing the data through a designated process. Other filesystem operations do not incur significant overhead.

5 Results

We have explored performance of the simple tasks such as array reduction and array exchange as well as performance of QD application as a whole.

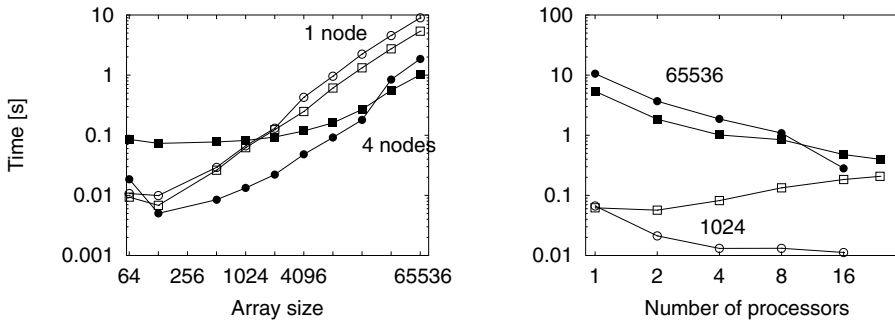


Fig. 1. Performance of array reduction for different array lengths and parallel execution as a function of processing elements. Pfortran results are denoted as squares and Co-Array results as circles.

The reduction algorithm execution times are given in Figure 1. In the single processor case, reduction simply consists of a sum over all vector elements. As in the array update, there is also a discontinuity at 4096 array elements in reductions using Co-Array Fortran and Fortran90. Recall that the reduction algorithm interprocess data exchange consists of a single scalar from each processor accumulating a processor's partial sum. Consequently, the cost to perform the reduction is dominated by the partial summation at each process. However,

interprocess communication costs become more apparent with short vectors as shown in figure 1. Interestingly, the Co-Array Fortran reduction algorithm, even though naively written as $\mathcal{O}(P)$, outperforms the $\mathcal{O}(\log P)$ Pfortran compiler generated algorithm. The difference is likely measuring *shmem* and *MPI*, underlying Co-Array Fortran and Pfortran, respectively.

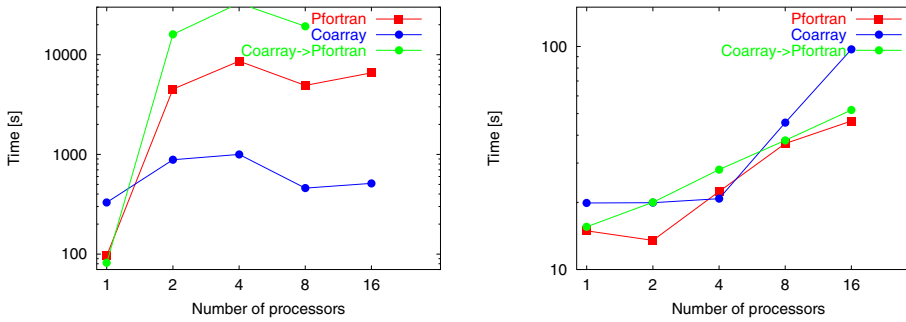


Fig. 2. Performance of the array exchange using short (left) and long (right) messages for different array length. Pfortran results are denoted as squares and Co-Array as circles. Full circles denote results for automatic translation of Co-Array code into Pfortran.

During QD code array exchange, the non-replicated data is sent to all processes in the order needed to obtain all data available at each processor. This task is performed using two different communication approaches. In the first, the arrays are sent element by element, which results in exchanging small portions of data. In the second model, communication is performed by a single exchange. In both communication approaches, the amount of exchanged data is the same therefore the of the exchange performance is dominated by the communication efficiency. Co-Array Fortran and Pfortran results in significant differences in performance due to the cost of communication initialization and process synchronization (Figure 2). The communication overhead is smaller while using Co-Array Fortran resulting in better performance for large numbers of short messages. Where large arrays are exchanged, Co-Array execution time increases almost linearly for number of processors grater than 2. The Pfortran code exhibits a slower ($O(log(P))$) increase of execution time with increasing numbers of processors as result of the underlying communication algorithm.

The overall performance for the 100 step propagation of quantum particle represented on the $32 \times 32 \times 32$ grid. presented in Figure 3 confirms high efficiency of parallelization. Both Pfortran and Co-Array codes scale linearly with the number of nodes, illustrating the viability of both parallelization tools. Small differences originate in differences in the performance of elementary array operations.

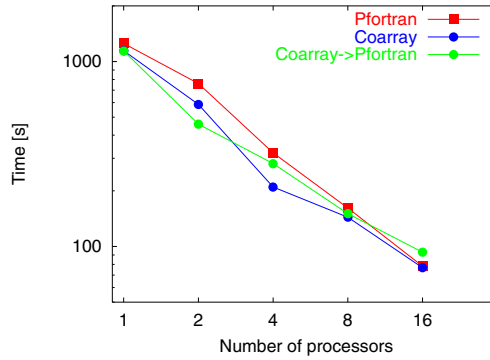


Fig. 3. Performance of the Quantum Dynamics code. Pfortran results are denoted as squares and Co-Array results as circles. Full circles denote results for automatic translation of Co-Array code into Pfortran.

6 Discussion

Our results show that efficient parallelization of large-scale scientific applications such as the QD code can be achieved using Pfortran and Cray's Co-Array Fortran.

The Pfortran and Co-Array Fortran implementations scale well with the number of processors, however, differences were observed in communication performance which results from the respective approaches to interprocess data movement. The small number of extensions and intuitive application of Pfortran and Co-Array Fortran are important considerations; HPF on the other hand, is more complex, resulting in code at times difficult to understand. We found the limited portability of Co-Array Fortran a disadvantage. Pfortran had performance comparable to Co-Array Fortran, and is without the portability limitations. In addition, the builtin Pfortran reduction operations along with the facility for user-defined ones are a definite plus for developing the Quantum Dynamics code. The Pfortran array exchange data reflects the communication algorithm used which results in logarithmic scaling. This could be implemented in Co-Array code, however this requires some additional programming effort.

In general, we found Co-Array Fortran and Pfortran both to be marked improvements over MPI for engineering parallel applications. In our opinion the two methods are complimentary paradigms, suitable for different algorithmic necessities. We plan to explore this concept further. The QD calculations are representative of a wide range of large-scale computational chemistry and scientific applications, suggesting general relevance of our findings concerning the efficacy of the parallelization tools.

Acknowledgements We thank Ridgway Scott for his comments and suggestions. Piotr Bala was supported by the Polish State Committee for Scientific

Research. Terry Clark was supported by the National Partnership for Advanced Computational Infrastructure, NPACI. The computations were performed using the Cray T3E at the ICM, Warsaw University, with Planguage compiler development performed in part at the San Diego Supercomputer Center.

References

1. H. Tel-Ezer and R. Kosloff. An accurate and efficient scheme for propagating the time dependent schroedinger equation. *J. Chem. Phys.*, 81:3967–3971, 1984.
2. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Compuiting*, 8, 1994.
3. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, and R. Manchek. PVM 3 User's Guide and Reference Manual. 1994.
4. A. Beguelin, J. Dongarra, G. A. Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Paralle Virtual Machine, A User's Guide and Tutorial for Networked Parallel Conmputing.* MIT Press, Cambridge, 1994.
5. R. Barriuso and A. Kniesi. Shmem User's Guide for Fortran. 1998.
6. R. W. Numrich. F^{--}: A Parallel Extension to Cray Fortran. *Scientific Programming*, 6(3):275–84, 1997.
7. R. W. Numrich, J. Reid, and K. Kim. Writing a multigrid solver using Co-Array Fortran. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Recent Advances in Applied Parallel Computing*, Lecture Notes in Computer Science 1541, pages 390–399. Springer-Verlag Berlin, 1998.
8. B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran: A parallel dialect of Fortran. *ACM Fortran Forum*, 11(3):20–31, 1992.
9. B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran (a parellel extension of fortran) reference manual uh/md-119. 1991.
10. P. Bala, P. Grochowski, B. Lesyng, and J. A. McCammon. Quantum–classical molecular dynamics. Models and applications. In M. Field, editor, *Quantum Mechanical Simulations Methods for Studying Biological Systems*, Springer-Verlag Berlin Heidelberg and Les Editions de Physique Les Ulis, 1996 pages 115–196.
11. T. N. Truong, J. J. Tanner, P. Bala, J. A. McCammon, D. J. Kouri, B. Lesyng, and D. Hoffman. A comparative study of time dependent quantum mechanical wavepacket evolution methods. *J. Chem. Phys.*, 96:2077–2084, 1992.
12. D. Kosloff and R. Kosloff. A Fourier method solution for the time dependent Schroedinger equation as a tool in molecular dynamics. *J. Comput. Phys.*, 52:35–53, 1983.
13. Bagheri Babak. *Parallel programming with guarded objects.* Research Report UH/MD, Dept. of Mathematics, University of Houston, 1994.
14. Robert J. Harrison. harrison@tcg.anl.gov, 1992.

Sparse Matrix Structure for Dynamic Parallelisation Efficiency

Markus Ast[1], Cristina Barrado[2], José Cela[2], Rolf Fischer[1], Jesús Labarta[2], Óscar Laborda[2], Hartmut Manz[1], and Uwe Schulz[1]

[1] INTES Ingenieurgesellschaft für technische Software mbH, Stuttgart, Germany
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. The simulated models and requirements of engineering programs like computational fluids dynamics and structural mechanics grow more rapidly than single processor performance. Automatic parallelisation seem to be the obvious approach for huge and historic packages like PERMAS. The approach is based on dynamic scheduling, which is more flexible than domain decomposition, is totally transparent to the end-user and shows good speedups because it is able to extract parallelism where others are not. In this paper we show the need of some preparatory steps on the big input matrices for good performance. We present a new approach for blocking that saves storage and decreases the computation critical path. Also a data distribution step is proposed that drives the dynamic scheduler decisions such that an efficient parallelisation can be achieved even on slow multiprocessor networks. A final and important step is the interleaving of the array blocks that are distributed to different processors. This step is essential to expose the parallelism to the scheduler.

1 Introduction

Although the increase of single processors performance, the requirements of engineering programs (like computational fluids dynamics and structural mechanics) for bigger and bigger models grows more rapidly. Simulations tend to require more accuracy, specify finer meshes, or increase the number of simulations. Standard models to deal with are around 1 million degrees of freedom (DoF) and up to 10 millions DoF for industrial benchmarks. For this reason, computational resources (like main memory limits or CPU time) are still a limiting factor in engineering. Out-of-core capabilities are essential to solve such problems.

Since scalability of single CPU becomes more and more difficult, the solution can not rely on computers speed alone. Parallelisation of the algorithms seem to be the obvious approach. The usage of parallel languages like HPF or new environments like Java can be a good strategy for new software [4.]. However, for huge and historic packages where rewriting would be too costly, the parallelism has to be integrated with incremental steps. Domain decomposition has been the a popular way to introduce parallelism in engineering packages. In this approach, the structure is divided into several meshes that can be solved in parallel and a last stage merges the results. Solvers based on domain decomposition show good speedups [10.] but they need more effort in the assembly

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 519-526, 2000.
© Springer-Verlag Berlin Heidelberg 2000

phase. Also the domain decomposition done for an architecture configuration usually is not appropriate for another. An alternative parallelisation strategy is applied in the PERMAS system. It is more flexible than domain decomposition because it can exploit domain grain parallelism but also finer grain parallelism [9.]. Moreover PERMAS parallelism is achieved automatically, thus, it is transparent to the programmer. In this way the whole code is parallelised (i.e. non-linear simulations or contact analysis), while others just have a parallel solver. Also PERMAS guarantees that the numerical results are independent of the number of processors used on their computation.

In this paper we evaluate how reordering and data distribution can improve the performance of PERMAS parallelisation. While the classical reordering step is applied to improve the matrix fill-in, new steps can be introduced before the actual computations in order to increase parallelism. We present the three new steps that PERMAS applies after the classical reordering step: blocking, data distribution and interleaving. The paper evaluates different heuristics and shows that the achieved speed-up is up to a 5.3 on a 8-processor SGI Origin 2000. As far as we know, other commercial out-of-core packages [7.] are only able to achieve 1.42 speedup on a 4-processor CRAY-YMP for a big problem and 3.26 speedup on 8-processor CRAY C90 for a small problem. Even the speedups of in-core parallel commercial systems [1.] are around 1.8 for small problems (from 4 to 180 thousands DoF). The paper is organized as follows. Section 2 presents the storage and parallelisation strategies adopted by PERMAS. Sections 3 and 4 detail the three preparatory steps (blocking, distribution, interleaving) and presents the measures of some simulations. Final remarks and conclusion are in section 5.

2 PERMAS Global Structure

The general purpose Finite Element (FE) system PERMAS is a commercial software with 20 years of history. Real problems of structural mechanics and fluid dynamics are the actual input data. These real problems are defined with an extremely large matrix (up to 10 millions degrees of freedom), called **hypermatrix**.

Storage. PERMAS stores the hypermatrix in a three levels structure. In the highest level, L3, we have the hypermatrix structure. Each element is either a pointer to a second level submatrix or null if all the elements of the submatrix are zero. Since the hypermatrix is symmetric, only the upper triangular of L3 is actually stored. In the second level, L2, we have again either indirections to L1 or null pointers. This two levels suppose only about a 5% of the total storage. In the last level, known as L1, we have the actual data. PERMAS maps the non-zero L1 blocks as dense arrays using the file system and handles their input/output to disk.

Parallelisation. The parallelisation strategy can be found in [2.] and here we just summarize the principal aspects. The PERMAS main module, which follows a loop-nested structure to traverse the L3 and L2 levels of the hypermatrix, generates a task for each numerical computation done over the L1 matrices. Each task is inserted on-the-fly in the Task Graph (TG). When the TG is larger than a threshold, the main module passes the control to an additional module, the **Parallel Task Manager** (PTM). The PTM con-

tains a dynamic scheduler and sends the ready tasks to slave processors (executors) using MPI. The executors do the numerical computations using standard BLAS calls. This strategy shows several advantages. Parallelisation is done automatic and transparent to the programmer because the program structure is the same for sequential and for parallel versions. Previous (sequential) PERMAS programs can be parallelised by just changing the BLAS calls with new PTM calls. The approach exploits a finer grain parallelisation than Domain Decomposition and thus, makes possible a better load balance. It is more flexible because a same executable works for different hardware configurations (number of processors) without recompilation. Finally, the numeric results are exactly the same for sequential or parallel (with any number of processors) because the operation dependences and the execution order do not change.

Preparatory Steps. It is well known that the parallel factorization can be improved with a preparatory re**ordering** step which permutes the nodes of the FE mesh. Besides the classical reordering, that PERMAS does using a combined technique of minimum degree and nested dissection [3.6.8.], it does three more preparatory steps: blocking, data distribution and interleaving: The **blocking** step consists on dividing the hypermatrix into its three level storage hierarchy. The intuitive way to do it is to superpose a grid on top of the hypermatrix twice: a fine grid defines the blocks of level L1 and a larger grid defines those of level L2. Section 3 presents an alternative algorithm for the L1 grid. New data structures are built after blocking to represent the matrix and the elimination tree at the L1 level. We call **Plane Array (PA)** to the matrix that represent L1 structure of A. Each element PA(i,j) represents a L1 block. A zero value means that the block is not really allocated. We named **Plane Elimination Tree** (**PET)** to the elimination tree of PA. These coarser-grain data structures are only needed on preparation. During execution the PTM exploits a task-level parallelism which is of a finer grain than the elimination tree parallelism [9.]. Next step is the **distribution**. It decides the initial assignment of the L1 blocks to processors. A good distribution is a compromise between a good load balance and a reduction of the communications. Section 4 presents the tight relation of the distribution and the PERMAS dynamic scheduling. It also evaluates several distribution alternatives and shows the need of the last preparatory step, interleaving.

3 Blocking: Fixed-Sized vs. Variable-Sized

The main objective of the hypermatrix blocking is the minimisation of the required storage, but, as we will show, this is not the only issue. Fig 1. shows the hypermatrix skyline of a motivating example, where the grey area represents the non zero elements after the reordering pass. Fig 1.a presents the classical blocking strategy of PERMAS, lets call it fixed-sized. The hypermatrix is divided into square blocks by superposing a grid on top of the it.

Fixed-sized blocking is a simple and clear strategy. Here, some tuning on the size of the blocks can help to minimise the storage requirements and the I/O overhead. This is an input parameter that usually ranges from 30x30 to 128x128 elements per block. There is a compromise between small sized blocking, which reduces the L1 stored zeros and

big sized blocking which reduces the number of blocks and thus minimises the I/O overhead. Fixed-sized blocking becomes a problem during parallel execution because of data dependences. The computations of L1 blocks (tasks) are subjected to the precedences of the PET. These precedences inhibit the dispatching of new computations. When the precedences are due to true dependences then they must be preserved. But precedences can be artificially created by blocking. These artificial dependences are not important on a sequential execution, but on a parallel execution they suppose longer critical paths and an increase of the computation time. These artificial dependences can disappear using variable-sized blocking.

Fig 1.Blocking alternatives

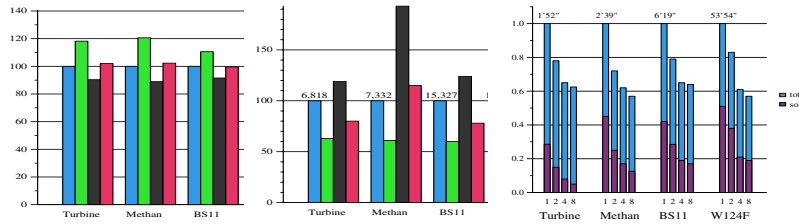
For example, let us consider the fourth diagonal block of Fig 1.a, PA(4,4). At the element level, there is a decoupling point that divides the block in two parts, let us call them $Up_{4,4}$ and $Low_{4,4}$. The computations of the elements of the two parts can be done in parallel. Moreover, all the elements of blockPA(4,5) can be also computed in parallel with the $Up_{4,4}$. Nevertheless, since the parallelism grain is the L1 level, the elements of the two parts of PA(4,4) belong to the same task and execute sequentialy. Moreover, the transitive closure of the dependences from PA(3,4) to PA(4,4) and from PA(4,4) to PA(4,5), creates the artificial dependence from PA(3,4) to PA(4,5). The variable-sized blocking proposed is illustrated in Fig 1.b. It finds the decoupling points of the hypermatrix and uses them as the vertices of the superposed virtual grid. The resulting blocks have different sizes and are not square. The variable-sized blocking decreases the number of dependences and moreover it saves disk area. On the other side, it increases the number of blocks. Fig 2. presents the results of simulating the solver part of 4 commercial benchmarks which characteristics are shown in Table 1. Numbers are given for two fixed-sized and two variable-sized blocking: Plot *fixed(16Kw)* stands for blocking into square blocks of 16Kwords (128Kbytes). Values are normalized to this first block-

Table 1. Benchmark description

Bench	Problem	DoF	total/solver time (m' s")	Mem (Mb)	I/O blocks
Turbine	Eigenvalue	66,456	1' 52" / 32"	90	9,278
Methan	ship structure	48,162	2' 39" / 1' 12"	135	8,978
BS11	rotating piece	111,057	6' 19" / 2' 38"	180	53,058
W124F	car	1,310,616	53' 54" / 27' 33"	810	209,940

ing. The plot *fixed(32Kw)* use the same strategy with double sized blocks. The two other plots, *variable(16Kw)* and *variable(32Kw)*, show the results of the variable-sized block-

ing when the L1 sizes are upper limited to 16Kwords or 32Kwords respectively. Fig 2.a shows the disk space needed to store the hypermatrices of the 4 benchmarks. Disk requirements are greater when blocks are bigger, because they include more zero-stored elements, while small and variable blocking covers better the shape of the hypermatrix with less space. Fig 2.b shows the number of blocks. This gives a measure of the dynamic overheads. When more (smaller) tasks are generated, more scheduling time and more input/output are expected. The Fig 2.c shows the expected execution time based on the critical path of the TG. The weight of all the tasks are considered to be equal to 1 when the block size was 16Kwords and equal to 2 for 32Kwords blocks (same for fixed than for variable sized -as the worst case-).



a) matrix size b) number of blocks c) critical path

Fig 2.Fixed vs. variable sized blocking

Looking at the simulation results we conclude that variable-sized blocking can save up to 10% of the disk storage. The new storage is more fragmented and introduces a 20% more overhead on the TGM and on I/O requests. Finally it reduces the critical path length on around 90%, thus, much more parallelism is exposed. Variable-sized blocking is now integrated in the PERMAS system as an option. CPU-time improvements are shown on most applications (i.e. 20% to 40% less execution time for Turbine with 16Kw block size).

4 Data Distribution and Interleaving

The data distribution main objective is to improve load balancing while minimising communications. Several algorithms [4.11.12.], based on recursive traverse of the elimination tree, have being proposed for column-based and submatrix based approaches (i.e. subtree recursive mapping of columns). They show benefits for the static parallelised solver. In this section we show the PERMAS approach. It does a preparatory step where the L1 blocks are assigned to virtual processors. Then, the dynamic scheduler [5.] uses this informations as a suggestion, but subjected to the availability of the actual processors.

Four different data distributions are tested for a Cholesky factorization on an 8 processor architecture. Fig 3.a shows their parallel speed-ups relative to the sequential execution and Fig 3.b shows their number of messages. We choose a 10Mbps (slow) Ethernet network as the worse case for message passing, in order to show that an efficient parallelisation is only possible with a good data distribution.The random, row-random and group-random distributions use an easy cyclic distribution with increasing

coarse levels. The random plot stands for a L1 block level distribution, while the row-random stands for a distribution done at the row level and group-random distributes groups of 10 rows. Since there are always dependences from the diagonal block to the rest of the blocks on the same row, the row-random distribution converts them to local and the number of messages decreases. For the architecture simulated this is still not enough to make the parallel execution faster then the sequential. The group-random distribution decrements more the number of inter-processor communications but still the speed-up is null. The last heuristic, named balanced, uses the PET to distribute

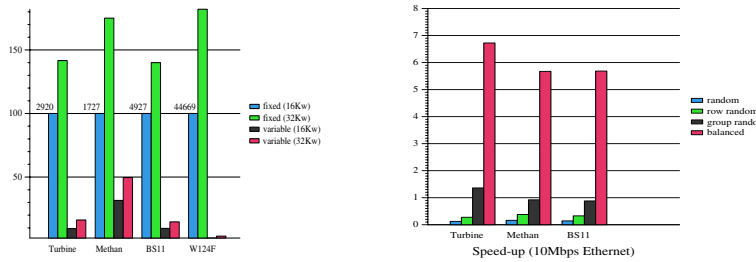
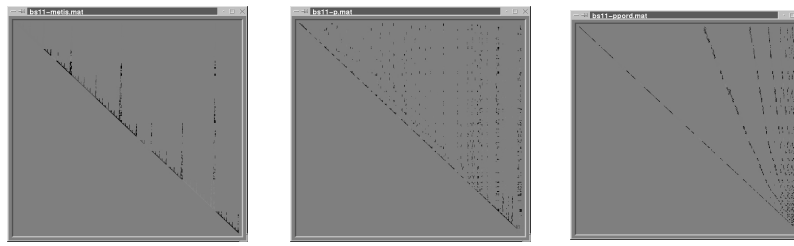


Fig 3.Data distribution simulations for a 8 multiprocessor

rows. Fig 3.b shows that this reduces the number of data communication again, now in a factor form 5 to 9. This reduction makes the difference in terms of speed-up, which is raised up to 6 for the simulated architecture. The balanced data distribution works as follows. Initially it assigns all the PET nodes to one processor. Then it enters in an iterative loop that decides to reassign a subtree from the most heavily loaded processor to the less loaded processor. The computational weight of the subtree is considered when deciding the PET cutting point. The loops iterates until a 5% threshold on processors balance is achieved.

The hypermatrix of Fig 4.a shows with 8 colours the results of the balanced distribution. The colours are clearly defined because joint consecutive rows. This block ordering is now a problem for the dynamic scheduler. The probability of having a Task Graph with tasks distributed to different processors is very low. The solution interleaving, this is, to find an equivalent reordering of the PA such that blocks distributed to a same processor are not consecutive.



a) data distribution b) interleaving by rows c) interleaving by blocks

Fig 4.Example of interleaving on BS11 for 8 processors

The mixture of colours of Fig 4.b an Fig 4.c shows this graphically. Such new reordering can expose the parallelism to the PTM from the beginning because the operands of the tasks on the dynamic TG are distributed over all the processors. Fig 4.b is achieved with a post-ordering at the block-row level. This schema showed very good speedups on the simulations but was no introduced in PERMAS environment because it introduced much storage in the L2 level. Fig 4.c shows the final heuristic integrated in PERMAS which uses a coarser post-ordering heuristic (10 rows).

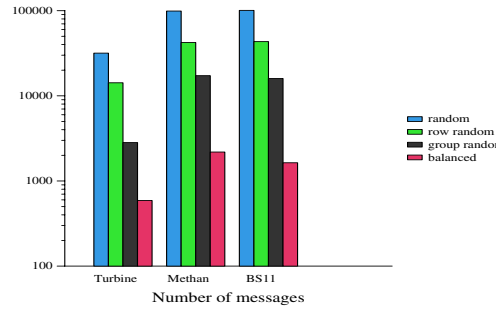


Fig 5.Elapsed time (SGI Origin 2000)

Finally, Fig 5. shows the performance of PERMAS parallelisation after applying the preparatory steps using 2, 4 and 8 slave processors. The total application execution time and the solver execution time are shown. Time savings of up to 20% and 40% of the total application time are achieved for 2 and 4 processors respectively. With 8 processors an additional gain of 5% saved time shows that more effort has still to be done in scalability. The main benefits are obtained on the parallelisation of the solver, but still the rest of the application has improvements from 10% to 15%. The solver speedup, that ranges from 2.4 to 5.3, is much better than speedups reported for Abaqus [1.] or MSC/Nastran [7.], which are less than 2 for large problems. This is an impressive performance if we consider the important amount of I/O overhead of the PERMAS out-of-core applications, specially in the backward and forward substitutions.

5 Conclusions and Future Work

This paper shows the need of several preparatory steps on sparse matrix structure for obtaining good performance of the automatic parallelisation of PERMAS. We propose a variable sized blocking of the hypermatrix and show how this blocking alternative saves storage and speeds the parallel execution. Also, a data distribution step is proposed and considered together with the dynamic scheduler that shows promising speedups even on slow multiprocessor networks. Finally, the interleaving step, done with a post-ordering algorithm, shows to be essential to expose dynamically the available parallelism. All these steps are integrated into the core of the PERMAS system. The speedups measured for real executions are much better than other commercial out-of-core FE systems. The benefits are achieved mostly for the solver part of the application, but PERMAS parallelisation approach also benefits the rest of the application. We are now working on the extension of the parallelisation to other parallel paradigms (multi

threading). We also plan to investigate additional parallelisation granularity (medium and coarse grain), and the parallelisation of all the application (matrix assembly operations, preparatory steps).

Acknowledgments. This work has been partially supported by the Ministry of Education of Spain under contract TIC98-0511, by the CEPBA and by the European Commission under ESPRIT contract n.22740 (PARMAT project).

References

1. Abaqus product performance. http://www.abaqus.com/products/p_performace58.htm
2. M. Ast, R. Fischer, J. Labarta and H. Manz. "Run-Time Parallelization of Large FEM Analyses with PERMAS". NASA'97 National Symposium. 1997.
3. T. Bui and C.Jones "A heuristic for reducing fill in sparse matrix factorization". 6th SIAM Conf. Parallel Processing for Scientific Computing, pp.445-452, 1993.
4. S. Fink , S. Baden and S. Kohn. "Efficient Run-Time Support for Irregular Block-Structured Applications". Journal of Parallel and Distributed Computing 50, pp.61-82. 1998.
5. T. Johnson. "A concurrent Dynamic Task Graph". International Conference on Parallel Processing, 1993.
6. G. Karypis and V. Kumar. "A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing. 1995.
7. L. Komzsik, "Parallel Processing in MSC/Nastran'. 1993 MSC World Users Conference, Virginia, 1993. http://www.macsch.com
8. V. Kumar et al. "Introduction to parallel Computing. Design and analysis of algorithms. The Benjamin/Cumminngs Pub. 1994.
9. J. Liu. "Computational models and task scheduling for parallel sparse Cholesky factorization". Parallel Computing 3, pp.327-342, 1986.
10. Marc product description. http://www.marc.com/Product/MARC
11. R. Schreiber. "Scalability of sparse direct solvers". Graph theory and sparse matrix computations, The IMA volumes in mathematics and its applications, vol. 56, pp.191-209, 1993.
12. S. Venugopal, V. Naik. "Effects of partitioning and scheduling sparse matrix factorization on communications and load balance". Supercomputing'91, pp.866-875, 1991.

A Multi-color Inverse Iteration for a High Performance Real Symmetric Eigensolver

Ken Naono[1], Yusaku Yamamoto[1], Mitsuyoshi Igai[2], Hiroyuki Hirayama[2], and Nobuhiro Ioki[3]

[1] Hitachi,Ltd., Central Research Laboratory
[2] Hitachi ULSI Corp.
[3] Hitachi,Ltd., Software Division

Abstract. An implementation of a real symmetric eigensolver on parallel nodes is described and evaluated. To achieve better performance in the inverse iteration part, a multi-color framework is introduced, in which the orders of the orthogonalizations are rescheduled so that the inverse iterations are executed concurrently. With the blocked tridiagonalization and backtransformation, our real symmetric eigensolver shows good performance and accuracy both on the MPP SR2201 and on the newly developed hybrid machine SR8000.

1 Introduction

In this paper, we treat an implementation of an eigensolver for dense real symmetric matrices that consists of tridiagonalization, bisection, inverse iteration and backtransformation. In each part, we adopted the existing algorithms, improved them from implementative point of views and produced an eigensolver of the matrix library MATRIX/MPP(03-00) for the hybrid[1] machine SR8000 [1]. For the tridiagonalization, the blocked method [2] [3] and lowering byte/flop techniques [4] were found to be successful. The bisection part can also be effectively parallelized [5]. When it comes to calculating a lot of eigenvectors, however, it is known that its parallel computation based on the conventional inverse iteration performs poorly because of the reorthogonalizations [6].

In 1997, Dhillon [7] proposed a new algorithm that solves each eigenvector in $O(N)$ time and produces automatically orthogonal eigenvectors without any reorthogonalization. The algorithm was implemented in the latest LAPACK(version 3.0) [8] subroutine dstegr, but the Dhillon's algorithm does not always work well when the relative gaps of eigenvalues are very small. In such cases, users have to use the conventional inverse iteration 'dstein' and endure poor performance. Furthermore, the ScaLAPACK [9] 'pdstein' allocates the clusters on the processing nodes, which usually results in the biased workload.

In this paper, we describe a new framework for the parallel computation of eigenvectors. Our framework, which we call a multi-color inverse iteration, was

[1] hybrid = combination of SMPs(symmetric multiple processors) and MPPs(massively parallel processors)

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 527–531, 2000.
© Springer-Verlag Berlin Heidelberg 2000

published locally in Japan [10]. It is based on the theory of the conventional inverse iteration with reorthogonalizations [11] [12]. One feature of our framework is that the orders of reorthogonalizations are rescheduled with coloring so that dependent eigenvalues are differently colored. Another is that the eigenvectors are evenly distributed over the nodes. Our framework enables some part of eigenvectors to be solved concurrently even though the reorthogonalizations are executed.

2 The Multi-color Inverse Iteration

The inverse iteration with reorthogonalizations is usually described as follows.

$$(T - e_i I)v_i^{k+1} = v_i^k, \qquad k = 0, 1, ..., \tag{1}$$

$$v_i^k := v_i^k - \sum_{j \in O_i} (v_i^k, v_j) v_j. \tag{2}$$

where T is a real symmetric tridiagonal matrix, I is the unit matrix, e_i is the i-th eigenvalue, v_i is the corresponding eigenvector and v_i^k is its k-th iterate. O_i denotes the indices set of eigenvectors against which v_i^k is reorthogonalized.

In the ScaLAPACK pdstein, the indices set O_i^{stein} is $\{\ j \in \mathcal{N}\ ;\ j < i,\ |e_j - e_i| < eps\ \}$, where 'eps' is the reorthogonalization criterion. So, all the eigenvalues in Fig.1, [2] for example, are gathered in one group and the eigenvectors are allocated in one node.

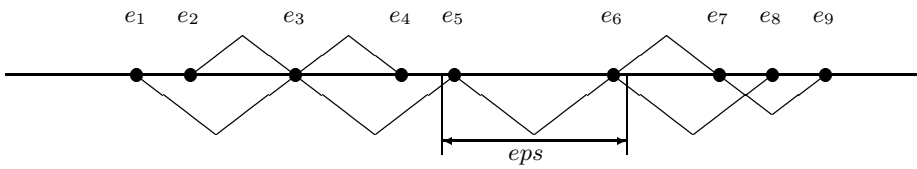


Fig. 1. The reorthogonalization criterion '*eps*' and 'connected' eigenvalues

In the multi-color inverse iteration, we find out the eigenvectors which can be solved independently. For that, we give colors to all the eigenvectors on conditions that connected eigenvectors should be colored differently. Table 1 gives a simple and easily implemented algorithm to color. The result is in the 0th stage of Fig.2. The calculations of eigenvectors with the same color have no data dependency with each other and can be done in parallel, because the eigenvectors need not be reorthogonalized with each other.

The colors also play a role of the priority of computation. First, eigenvectors with $color(i) = 1$ are calculated, second, those with $color(i) = 2$, and so on. The indices set O_i^{multi} is $\{\ j \in \mathcal{N}\ ;\ |e_j - e_i| < eps,\ color(j) < color(i)\ \}$.

[2] The eigenvalues are connected with polygonal lines when the distances between them are less than the reorthogonalization criterion 'eps'.

The eigenvectors are evenly distributed among nodes as in Fig.2. If necessary, each node receive the calculated eigenvectors from other nodes by internodes communication. In the second stage, for example, v_6 is transferred from N1 to N2, and v_7 is calculated by the inverse iteration with reorthogonalizations against v_6 and v_9. Thus the multi-color framework enables effective parallel implementation that is summerized in Table 2.

Table 1. A greedy algorithm for coloring eigenvalues ($e_i \leq e_j$ for $i < j$)

1. Set $color(1) = 1$ and $i = 2$.
2. Set $color(i)$ to a natural number satisfying the following two conditions.
   - As small as possible.
   - For any j with $0 \leq e_i - e_j < eps$, $color(i) \neq color(j)$.
3. $i = i + 1$, and if $i \leq n$ goto 2, otherwise stop.

coloring accomplished (0th stage)

node	N0			N1			N2		
eigenvector	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$color = 1$	○			○		○			○
$color = 2$		○			○		○		
$color = 3$			○					○	

1st stage

node	N0			N1			N2		
eigenvalue	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$color = 1$	◎			◎		◎			◎
$color = 2$		○			○		○		
$color = 3$			○					○	

2nd stage

node	N0			N1			N2		
eigenvector	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$color = 1$	○			○		○			○
$color = 2$		◎			◎		◎		
$color = 3$			○					○	

3rd stage

node	N0			N1			N2		
eigenvector	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
$color = 1$	○			○		○			○
$color = 2$		○			○		○		
$color = 3$			◎					◎	

Fig. 2. Allocation and the multi-color inverse iteration procedure

3 Numerical Tests and Remarks

First, on one node of the SR2201 [3], we compare the residual, orthogonality and time of the multi-color implementation with those of the equivalence of the LAPACK dstein for the [1,2,1] matrix with dimension 2000. The orthogonality is scaled as $||V^T V - I||_F, V_{ij} = (v_i, v_j)$ and the reorthogonality criterion '*eps*' changed from 10^{-6} to 1.0. The result in Table 3 shows that the multi-color inverse iteration performs better and the residual and orthogonality are almost same as those of the equivalence of dstein.

[3] The SR2201 has 300Mflop/s per 1 node and 300MB/s internode bandwidth.

Table 2. A parallel implementation of the multi-color framework

1. Calculate the indices set O_i^{multi} for all i on all nodes.
2. Define the $color(i)$ for all i on all nodes.
3. Do k = 1, Total_Color_Number
   Do i = 1, Total_Eigenvector_Number
   IF (Color(i)=k and My_Node_Number=Eigenvec_Alloc(i))
   (a) Get $v_j \in O_i^{multi}$ from other nodes if necessary.
   (b) Do the inverse iteration performing the following stages alternately.
      i. Solve the linear equation (1).
      ii. Reorthogonalize as in the equation (2) with O_i^{multi}.

Second, we evaluate scalability of the multi-color implementation on the SR2201 for the [1,2,1] matrix with dimension 8000. The result in Table 4 shows that scalability is low and there is room for improvement, especially in eps=1.0e-2. But note that, with the ScaLAPACK pdstein, all eigenvectors in that case fall into one group and no parallelism would be achieved.

We also evaluate the performance and accuracy for a real symmetric eigensolver [4] with the multi-color inverse iteration on the SR8000[4] . Test matrices are the Frank matrix $A_{ij} = min(i, j)$ with the dimension 8000 and 16000. The reorthogonalization criterion used is 10^{-5}. The execution time of each part, and total accuracy(r:residual, o:orthogonality) and performance are shown in Table 5. In both dimensions, the multi-color inverse iteration is confirmed to scale well, and total high performance is achieved.

However, we have to prove rigorously for rescheduling and test on a lot of clustered matrices, which will be our future work. Adoption of 'dtwqds [7]' will be important to solve the low scalability problem.

References

1. SR8000 HOMEPAGE: http://www.hitachi.co.jp/Prod/comp/hpc/eng/sr81e.html
2. J. J. Dongarra and R. A. van de Geijn: Reduction to condensed form for the Eigenvalue problem on distributed architectures, Parallel Computing Vol. 18, No. 9, pp. 973-982, 1992.
3. H. Y. Chang, S. Utku, M. Salama and D. Rapp: A parallel Householder tridiagonalization stratagem using scattered square decomposition, Parallel Computing Vol. 6, No. 3, pp. 297-311, 1988.
4. K. Naono, Y. Yamamoto, M. Igai, H. Hirayama: High performance implementation of tridiagonalization on the SR8000, Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC-ASIA2000), Beijing, China, pp.206-219, 2000.

[4] The SR8000 has 8Gflop/s per 1 node and 1GB/s internode bandwidth.

Table 3. Residual, orthogonality, and performance on 1 node of the SR2201

eps		1e-6	1e-5	1e-4	1e-3	1e-2	1e-1	1e-0
res	multi-color	4.2e-14	4.2e-14	4.2e-14	4.2e-14	4.2e-14	4.2e-14	3.8e-14
	equi-dstein	4.2e-14	4.2e-14	4.2e-14	4.2e-14	4.2e-14	4.1e-14	3.5e-14
ortho	multi-color	4.5e-11	4.5e-11	1.4e-11	4.2e-12	9.7e-13	2.7e-13	8.3e-14
	equi-dstein	4.9e-11	4.9e-11	1.4e-11	4.3e-12	9.2e-13	2.5e-14	8.2e-14
time	multi-color	3.73s	3.75s	3.75s	3.90s	5.67s	19.9s	151.2s
	equi-dstein	5.19s	5.20s	5.22s	5.47s	9.31s	36.4s	211.5s

Table 4. Execution time and speedup rate (in brackets) on the SR2201

eps	4 nodes	8 nodes	16 nodes	32 nodes	64 nodes
1.0e-2	85.8 s (1.00)	70.6 s (1.22)	59.3 s (1.45)	55.0 s (1.56)	55.0 s (1.56)
1.0e-4	16.3 s (1.00)	9.4 s (1.73)	6.2 s (2.63)	6.5 s (2.50)	5.8 s (2.81)
1.0e-6	15.9 s (1.00)	9.0 s (1.77)	5.7 s (2.80)	4.4 s (3.61)	4.0 s (4.00)

(When 4 nodes, the speedup rate is 1.0.)

Table 5. Execution time and accuracy result for Frank matrices on the SR8000

No. of nodes		1	4	16
N=8000	total	440.2 s	120.6 s	46.47 s
acu.	trid.	130.0 s	41.4 s	20.9 s
	bisec.	60.6 s	15.2 s	3.9 s
r:1.47e-8	m-inv.	87.7 s	22.3 s	9.3 s
o:1.04e-10	back.	161.9 s	41.8 s	12.2 s

No. of nodes		1	4	16
N=16000	total	2857.4 s	740.0 s	227.5 s
acu.	trid.	983.1 s	266.5 s	89.9 s
	bisec.	242.3 s	60.6 s	15.2 s
r:1.38e-7	m-inv.	354.1 s	90.1 s	34.0 s
o:2.62e-10	back.	1277.8 s	322.7 s	88.4 s

5. J. Demmel, I. Dhillon, and H. Ren: On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic, Electronic Trans. Numer. Anal. 3, 116-149, Dec. 1995.
6. J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R.C. Whaley : LAPACK Working Note 95, ScaLAPACK: A Potable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance, 1995.
7. I. S. Dhillon: A New O(n^2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem, Ph.D. thesis, Computer Science Division, University of California, Berkeley, May 1997.
8. LAPACK: http://www.netlib.org/lapack
9. ScaLAPACK: http://www.netlib.org/scalapack
10. K. Naono, M. Igai, Y. Yamamoto: Development of a Parallel Eigensolver and its Evaluation, Proceedings of the Joint Symposium on Parallel Processing 1996, Waseda, Japan, pp9-16, 1996(in Japanese).
11. G. Peters and J.H. Wilkinson : The Calculation of Specified Eigenvectors by Inverse Iteration pp.418-439, in book 'Linear Algebra' edited by J.H.Wilkinson and C.Reinsch, Springer-Verlag, 1971.
12. B. Parlett : The symmetric Eigenvalue Problem, Prentice Hall, Englewood Cliffs, NJ, 1980.

Parallel Implementation of Fast Hartley Transform (FHT) in Multiprocessor Systems

Felicia Ionescu, Andrei Jalba, Mihail Ionescu

University "Politehnica" Bucharest,
Str. Iuliu Maniu Nr. 1-3, Bucharest, Romania
{fionescu, andrei, mihail}@atm.neuro.pub.ro

Abstract. The purpose of this paper is to investigate the parallelization of one-dimensional Fast Hartley Transform (FHT) algorithm on shared memory multiprocessor systems. The computational dependencies of the sequential FHT algorithm are analyzed, in order to distribute the loops of the algorithm among multiple processes (threads), executed on the available processors of the system. The outer loop of the algorithm carries data dependencies between consecutive iterations and, for parallel execution, synchronization barriers are introduced. The results show that in the parallel execution of the FHT algorithm a significant speed-up is obtained and that the speed-up increases with the size of the input sequence.

1 Introduction

Discrete Hartley Transform (DHT), like *Discrete Fourier Transform* (DFT), plays an important role in digital signal processing. DFT is very used, but it includes complex arithmetic even if the input sequence is a real one. Hence, DHT was developed to eliminate this redundancy for a real sequence of numbers. The DHT of a sequence of N real numbers X(i), i = 0,1,....,N – 1, is:

$$H(k) = \sum_{i=0}^{N-1} \left(X(i) \left[\cos(2\pi ki / N) + \sin(2\pi ki / N) \right] \right). \quad (1)$$

The computational complexities of both transformations, DFT and DHT, are in $O(N^2)$. As well as DFT, DHT has a fast version called *Fast Hartley Transform* (FHT) [1], with the time complexity in $O(N \log_2 N)$. The purpose of this paper is to investigate the parallelization of FHT algorithm on shared memory multiprocessors.

2 The Analysis of the Sequential FHT Algorithm

Several different forms of the FHT algorithm exist, but in this paper we use for parallelization a radix-2 decimation-in-time FHT algorithm [2]. Fig. 1 illustrates the computational flow graph for N = 8-points FHT algorithm.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 532-535, 2000.
© Springer-Verlag Berlin Heidelberg 2000

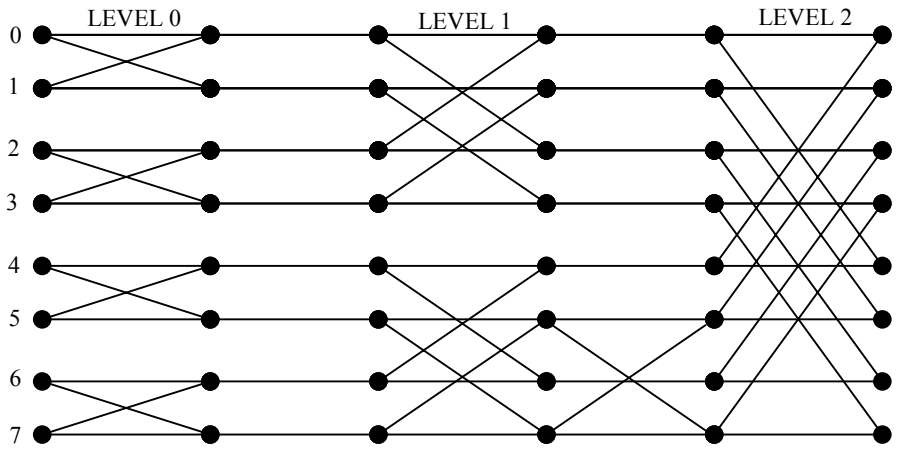


Fig.1. Computational flow graph for the 8-points FHT

An examination of Fig. 1 reveals that FHT computations at each level resemble basic FFT butterfly computations. The first level (r = 0) consists of 2-points FFT-like butterflies and the remaining levels (r = 1,2,...n – 1, where $n = \log_2 N$) consist of 4-points FHT butterflies. There are two types of FHT butterflies, which will be referred here as T1 and T2 4-points basic butterflies. Each type of FHT butterflies, which are presented in Fig. 2, is identified by a 4-tuple (p, r, q, s).

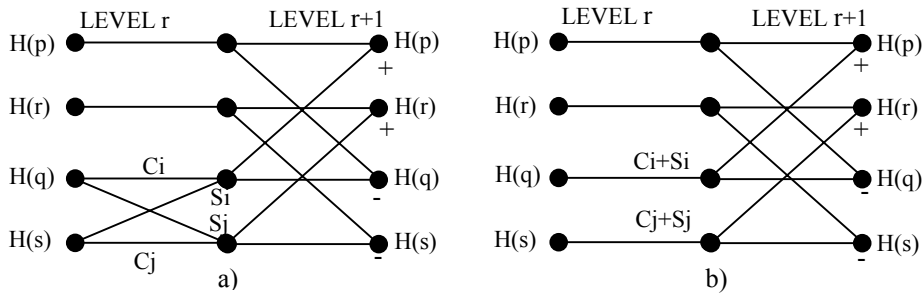


Fig. 2. Computational flow graphs for a) T1 and b) T2 4-points butterflies. Ci = cos (2πi / N), Si = sin (2πi / N)

The C-like pseudo-code of the sequential FHT algorithm is given bellow.

```
/* Sequential FHT algorithm
   Input in bit-reversed order in H[0…N-1]
   Output in normal order in H[0…N-1] */
for(i = 0; i < N/2; i++){
  temp = H[2i+1];
  H[2i+1] = H[2i] - temp;
  H[2i] = H[2i] + temp;}
```

```
for(r = 1; r < n; r++) {
  for(i = 0; i < N/pow(2,r+1); i++) {
    p2 = i*pow(2,r+1); q2 = p2 + pow(2,r);
    r2 = p2 + pow(2,r-1); s2 = q2 + pow(2,r-1);
    tmp1 = H[q2]; tmp2 = H[s2];
    H[q2] = H[p2] - tmp1; H[s2] = H[r2] - tmp2;
    H[p2] = H[p2] + tmp1; H[r2] = H[r2] + tmp2;
    for(j = 1; j < pow(2,r-1); j++) {
      p1 = p2 + j; q1 = p1 + pow(2,r);
      r1 = p2 + pow(2,r) - j; s1 = r1 + pow(2,r);
      tmp1 = Ci*H[q1] + Si*H[s1];
      tmp2 = Cj*H[s1] + Sj*H[q1];
      H[q1] = H[p1] - tmp1; H[s1] = H[r1] - tmp2;
      H[p1] = H[p1] + tmp1; H[r1] = H[r1] + tmp2;
    }
  }
}
```

As it is shown above, the first *for* loop performs computations required by the first level (r = 0), corresponding to the 2-points butterflies. The second outer *for* loop performs the computations for the remaining n – 1 levels. The first inner *for* loop iterates $N/2^{r+1}$ times to compute the T2 butterflies at each level r and the innermost *for* loop iterates $2^{r-1}-1$ times to compute the T1 butterflies. The numbers of T1 and T2 butterflies at level r are:

$$S_{T1}(r) = N/4 - N/2^{r+1}\ ;\ S_{T2}(r) = N/2^{r+1}\ . \qquad \textbf{(2)}$$

3 Parallelization of the FHT Algorithm

For parallelization of N-points FHT on a shared memory multiprocessor system, a number of P threads run on P processors of the system, executing the code that implements the parallel version of the algorithm. The only computational dependency in the sequential version of the algorithm exists between successive levels because level-r computations depends on the level-r-1 results. For this reason, we distribute the iterations of each level loop and provide synchronization mechanisms between threads (implemented using barriers) at the beginning of each level. These barriers are needed before the beginning of the second outer *for* loop that performs the computations for levels 1,2,.... n –1 and before the first inner *for* loop that iterates $N/2^{r+1}$ times to compute the T2 butterflies.

Because the number of iterations of the *for* loops which compute the T1 and T2 butterflies are dynamically varying as a function of r (the current level), we have chosen to parallelize the *for* loop with the greatest number of iterations. The level r for which the number of iterations of these two *for* loops is the same is obtained by equaling the numbers of butterflies of type T1 and T2 given by (2) and has the value r = 2. For r ≤ 2 the loop corresponding to T2 butterflies is parallelized; for r > 2 the

loop corresponding to T1 butterflies is parallelized; in this case, the computations for the T2 butterflies are done only by one process (thread).

The pseudo-code of the parallel version of the algorithm is presented bellow.

```
/* Parallel FHT Algorithm */
forall(0 ≤ p ≤ P-1) {
  for(j = p*N/(2P); j < (p+1)*N/(2P); j++) {
    temp = H[2j+1]; H[2j+1] = H[2j] - temp;
    H[2j]= H[2j] + temp;}
}
barrier synchronization;
for(r = 1; r < n; r++) {
  barrier synchronization;
  if(r ≤ 2)
    forall(0 ≤ p ≤ P-1) {
      for(i = p*N/(P*pow(2,r+1));
          i < (p+1)*N/(P*pow(2,r+1)); i++) {
        Compute T2 butterfly;
        for(j = 1; j < pow(2,r-1); j++)
          Compute T1 butterfly;
      }
    }
  else
    for(i = 0; i < N/pow(2,r+1); i++) {
      Compute T2 butterfly;
      forall(0 ≤ p ≤ P-1) {
        for(j = p*pow(r-1)/P;
            j < (p+1)*pow(2,r-1); j++)
          Compute T1 butterfly;
      }
    }
}
```

4 Results and Conclusions

For evaluation of performances, the parallel FHT algorithm was implemented on a two-processor IBM RS/6000 station, under AIX 4.3 operating system. The initial data are arrays of different dimensions and their elements are double-precision real numbers. The results show that the execution speed of the parallel algorithm can be increased using all processors in a multiprocessor system and that the speed-up increases with the size of the input sequence.

References

1. Bracewell, R.N.: The Fast Hartley Transform. Proceedings of IEEE, Vol. 72 (1984) 124-132
2. Aykanat, C., Dervis, A.: Efficient Fast Hartley Transform Algorithms for Hypercube-Connected Multicomputers. IEEE Transactions on Parallel and Distributed Systems, Vol. 6, (1995) 561-577

Topic 08
Parallel Computer Architecture

Silvia Müller, Per Stenström, Mateo Valero, and Stamatis Vassiliadis

Topic Chairpersons

Computer architecture is a truly fascinating field in that improvements in the basic technology and innovations how to make best use of the underlying technology has yielded a performance growth exceeding a million times over the past 50 years. What is even more amazing is the fact that the pressure on maintaining this rate of performance growth shows no decline. In fact, as performance thresholds are passed, application designers face new opportunities that give new challenging problems to work on for computer architects.

Parallelism and locality are the two fundamental concepts from an architecture point of view that have contributed to the impressive performance growth. Exploitation of parallelism has led to an increased pressure on the memory system. The increased speed-gap between processor and memory has in turn fueled innovations in memory hierarchy research that exploit locality.

Until now, the major form of parallelism that has been exploited at the microprocessor level is across instructions. Coarser-grained, or thread-level parallelism, is becoming increasingly important to consider for the following two reasons: First, there are always computational problems at any one time whose performance demands can not be accommodated by a single processor, such as various forms of transaction and database processing and scientific/engineering computing. Second, exploiting instruction-level parallelism is yielding diminishing returns owing to the complexity involved in considering larger instruction windows. Both of these observations prompt towards also exploiting thread-level parallelism and are the major motivating factors for parallel computer architecture – the topic of this session.

Thread-level parallelism can be exploited at the chip as well as at the system level. Two architectural styles at the chip level are currently being debated: chip multiprocessors and multithreaded architectures. Independent of the architectural style chosen at the chip level, how thread-level parallelism is exploited across microprocessor chips, which act as processing nodes, is an important issue in the area of parallel computer architecture.

Historically, message passing (or distributed memory) and shared-memory multiprocessors are the prevailing parallel computer architectural styles at the system level. In message passing, the software abstraction forces threads to explicitly exchange messages between disjoint address spaces whereas in shared-memory threads exchange messages implicitly in a common address space. In implementing any of these abstractions, a fundamental issue is to reduce the impact of inter-thread message communication latency on the execution time of parallel programs. All the papers in this session address in one or another way

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 537–538, 2000.
© Springer-Verlag Berlin Heidelberg 2000

this fundamental problem by either proposing innovative solutions to reduce or tolerate the message latency, or by making important observations regarding the nature of the inter-thread communication pattern in parallel programs to be used to identify new approaches for efficient communication.

In shared-memory multiprocessors inter-thread communication results in coherency interactions between threads that may hurt performance. In the first paper, Acquaviva and Jalby study the nature of these interactions based on analysis of a suite of scientific codes and make interesting observations regarding what program behavior causes performance problems. These observations are important in order to find more efficient coherency mechanisms.

In message-passing systems, the thread that sends the message often has to wait until the receiver has copied the data into its address space. One interesting contribution in the second paper by May et al. is the introduction of a new message passing protocol that allows the sender to copy the data directly into the address space of the receiver.

Replication of data is an effective means to avoid some of the communication in shared-memory machines and the COMA concept enables replication also at the memory level. In the third paper, Ferraris et al. use the COMA concept to propose a multiprocessor architecture using workstations as building blocks. They report on a COMA protocol that reduces the overhead associated with replication.

For small-scale systems, bus-based multiprocessors have dominated the market for some time and are also considered for chip multiprocessors. A problem with these systems is that inter-thread communication can cause severe bus contention. In the fourth paper, Milenkovic and Milutinovic propose an innovative solution, called cache injection, to reduce the bus traffic.

Finally, the topic of the last paper by Talbot and Kelly is again on replication at the memory level in cache-coherent NUMA machines. In these machines, widely shared memory blocks can cause performance problems if the cache space is not sufficient. In their proposal, called adaptive proxies, a mechanism is proposed that adaptively replicates only the data that is simultaneously shared by a large number of nodes.

We hope you will enjoy and learn a lot from this collection of papers.

Coherency Behavior on DSM: A Case Study

Jean-Thomas Acquaviva[1] and William Jalby[2]

[1] CEA/DAM French Atomic Energy Commission,
acquaviv@bruyeres.cea.fr,
[2] PR*i*SM Lab. Versailles University,
jalby@prism.uvsq.fr

Abstract. This paper summarizes a characterization effort of coherency traffic in shared memory scientific applications. In particular, based on a systematic experimentation study of the well known Splash 2 benchmarks, two properties are detailed: locality of coherency activity within data set and within application code.
Characterizing properly these properties is essential for both restructuring applications to improve coherency behavior and/or design new cost effective coherency mechanisms. Consequently, as a result of our analysis, from the exposed fact that data balance between two strongly marked behaviors and that a small fraction of application code is responsible of the majority of coherency traffic, we propose various research directions for improving performance of coherency actions.

1 Introduction

Nowadays even if the basics of coherency mechanism design are well understood, with many proposals (IBM, SUN, HP, SEQUENT, SGI), performance of these mechanisms is still a major issue. Coherency optimizations have been and still are a very hot research topic. Numerous optimization mechanisms (mostly hardware) have been proposed and evaluated [7], [4], [3]. Most of the time, the evaluation strategy reveals clear relation between the proposed mechanism and some key characteristics of the application [2], [6]. The number of studies to characterize coherency behavior of applications is still fairly limited [9], [8] [1] and the phenomenon is still not well understood. Such a knowledge is important to bring further good optimization schemes addressing the real problem.

This paper addresses the coherency traffic characterization problem for scientific applications on DSM. From our set of experiments, two properties are analyzed, a more complete version of our work being presented in [11].

In figure 1, coherency traffic is plotted in a two dimensional space: x-axis represents time while y-axis corresponds to cache line numbers. This traffic appears to be highly structured, exhibiting burst accesses, as well as cyclic patterns, presence of regular streams or hot memory regions. It should be noted that coherency events are highly clustered in time and in pace, making the use of average values extremely difficult to handle properly. This regularity can be an

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 539–544, 2000.
© Springer-Verlag Berlin Heidelberg 2000

asset for well tuned optimization mechanisms. We are convinced that the structure of coherency traffic takes its root in the intrinsic properties of application code.

Our work is complementary to the work done in [9], [8], [1]: different metrics are reported. Our work might also be of special interest for software optimization schemes, such as those exposed in [5], because we studied in depth correlations between source code and run time behavior.

The remainder of the paper is organized as follows, section 2 details the followed framework and depicts the experimental environment. Section 3 investigates aspects related to data activity, and section 4 analyzes code activity. Concluding remarks and openings are given in section 5.

2 Framework / Experimental Set-Up

Simulated Architecture: Parallelism is expressed at the loop level via fork and join (SPMD model). Consistency model supported is relaxed consistency. Using the Prism [1] [10] execution-driven simulator we model a DSM system, with a S-COMA memory management scheme. The simulated architecture consists of 8 single CPU nodes, each of them includes a PowerPC 604e with 4MB of L2 cache, a network interface and a hardware protocol engine. Coherency is maintained at the cache line granularity (set to 64 B).

Benchmarks Analysis: Benchmarks suite is composed of Splash, Splash-2 codes and CG (with 2 different conditioners) coming from NAS. According to their complexity, these codes can be simply classified as kernels for LU, FFT, Radix, simple codes for CG-Dia and CG-Poly, and complex codes for MP3D, ocean both contiguous and non contiguous, and Water-Spatial. Due to the lack of space, figures are given only for one representative benchmarks of each category.

From execution detailed log files are collected and during the post-processing stage these traces are injected in a Postgress database [2]. Resorting to a database allows us to track correlations and to generate statistics from various angles of investigation quickly and in a very efficient manner.

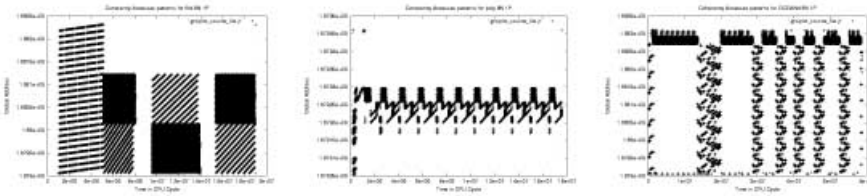
3 Data Activity

The data activity of a cache line is defined as the total number coherency events hitting that given cache line during the whole program execution.

Figure 2 is a cumulative representation of the fraction of coherency events related to the fraction of memory area. If coherency events were evenly distributed among cache lines, the resulting graph would be a diagonal. A steep slope on these figures correspond to a high disparity of coherency events toward cache

[1] Thanks to Ekanadham Kattamuri from IBM for his helpful advises and comments on the simulator

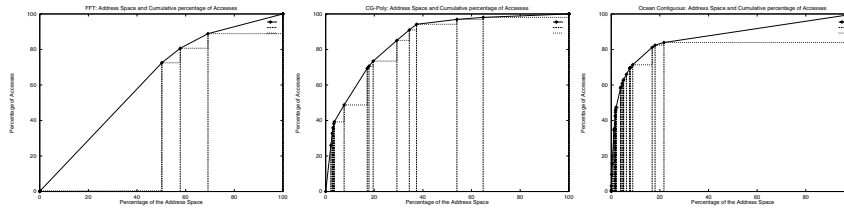
[2] Thanks to Gregory Watts from LRI lab. (Orsay University) for his help on Postgress database.



FFT CG-Poly Ocean-Contiguous

Fig. 1. Coherency Traffic: A highly structured phenomenon.

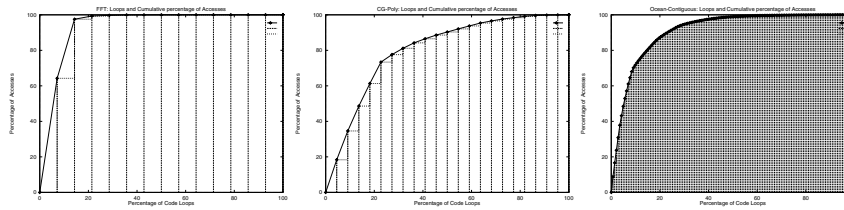
Each plot exposes shared memory accesses over execution time for an application. X-axis is the execution time, and Y-axis represents shared memory address space. Every dot plotted on the figure at *(x,y)* corresponds to coherency event occurring at time x aiming at cache line numbered y. Due to scaling and resolution problems, areas of intense activity appear as uniformly dark.



FFT CG-Poly Ocean-Contiguous

Fig. 2. *Correlation between cache lines and coherency events.*

With each cache line a weight is associated, computed as the number of coherency events aiming at this cache line divided the total number of coherency events. Cache lines are sorted on a decreasing weight basis. Cache lines with the same weight are coalesced in a memory region, such homogeneous memory regions appear as boxes on the figure. X-axis represents the percentage of the memory space. Y-axis is the cumulative weight represented by these cache lines. For instance, a dot located at *(x,y)* means that x % of the heaviest cache lines summarize y % of the total number of coherency events.



FFT CG-Poly Ocean-Contiguous

Fig. 3. *Correlation between loops and number of coherency events.*

With each loop a weight is associated, computed as the number of coherency events triggered within this loop over the total number of coherency events. Loops are sorted relatively to their decreasing weight. X-axis represents the percentage of active loops, i.e. generating coherency events, within the code. Y-axis is the cumulative weight represented by these loops. A dot located at *(x, y)* means that x % of the heaviest loops summarize y % of the total number of coherency events.

lines: a limited number of cache lines concentrates a lot of coherency events. In the same way, a curve reaching a plateau reveals a cold memory region, where the following cache lines summarize a negligible part of coherency events.
Another aspect to investigate is memory disparity behavior, cache line with a similar number of coherency events are gathered in a memory region. Boxes on figure 2 depict these regions. Clearly, on each plot a large box, corresponding to cache lines hit once, count for a small fraction of accesses. We call the corresponding memory space a cold region. From figure 2 three conclusions can be drawn:
Memory Disparity, coherency events are not dispatched on a homogeneous manner on memory. Radix, Lu and FFT are very regular. The breakdown of the memory space depending the number of accesses per cache line is composed of a limited number of large blocks. Memory space in Ocean both Contiguous and non-Contiguous, Water-Spatial and MP3D is decomposed in much more blocks.
Cold Regions, on every benchmark a large part of the memory space gathers only a limited fraction of coherency events. This is particularly obvious in Ocean-Contiguous, CG-Dia and MP3D. Where respectively 78 %, 54 % and 44 % summarize 14 %, 1 % and 3 % of coherency events. In the opposite, Radix, LU and FFT are near linear, this is induced by the limited number of epochs in these codes, and access strongly dominated by cold-start effects.
Hot Regions, symmetrically, a few cache lines concentrates activity. For Ocean-Contiguous, 5 % of the memory space concentrates 63 % of coherency events. In water-Spatial, 10 % of cache lines summarize 55 % of coherency events. MP3D has 52.5 % of coherency events targeting less then 8.5 % of the memory space. Obviously code with linear behavior (FFT, Radix and LU) do not present hot memory regions.
Research Tracks: An issue for cost-effective optimization mechanisms is to detect memory disparity. Sorting cache lines, statically (i.e. compiler task) or dynamically, on the basis of their coherency activity opens, at least, two perspectives: the first axis is to focus on hot cache lines, complex anticipation schemes, resorting on predictor or history, only need to track these hot cache lines. The second point is to provide a minimal coherency support for cold cache lines.
Many of proposed optimizations are relatively costly, for instance Mukherjee and Hill [4], and Lai and Falsafi [3] attach a coherency predictors to each cache line, by limiting their usage to hot region, we could reduce their cost drastically.

4 Code Activity

Code activity is defined for each loop of the code as the number of coherency events it triggers. Figure 3 is a cumulative representation of the fraction of coherency events generated by a fraction of code loops. The plot follows the same model than figure 2. From the figure 3, as with data activity, we observe that a limited amount of loops yields to the majority of coherency events. Even with an aggressive threshold of 80 %, approximatively 20 % of the loops are sufficient to catch the main part of the coherency traffic.

Research Tracks: These results illustrate that loop level is the good granularity level to observe coherency traffic, furthermore few loops drive a major part of the traffic. Detecting these hot loops, statically or even dynamically will lead to better traffic anticipation. Compiler efforts can be focused to optimize only a small part of the code. We believe that important step toward coherency bursts anticipation will be made with hot loops detection / marking schemes.

5 Conclusion and Future Works

This paper describes a part of our research on coherency traffic characterization, which is essential for optimizing coherency in a DSM architecture. In extension to previous studies, this work aims at depicting several key properties either corroborating these previous studies or bringing new facts to light.

From this characterization two directions for future works appear as promising. The first direction is to enhance the post-processing stage. Datamining is a very appealing technique. Instead of generating by ourselves sets of complex queries, a high potential is contained within automated datamining for correlation detection. This will allow to prospect for more metrics, and constitute a step toward genericity. Exploitation of application characteristics is the second direction for further reseaerch. A natural way to pursue our work is to investigate an optimization scheme, which can be purely hardware or also enjoying compiler support.

References

1. G. Abandah and E. Davidson. Configuration Independent Analysis for Characterizing Shared-Memory Applications. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS'98)*, March 1998.
2. John Carter, John Bennett, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.
3. An-Chow Lai and Babak Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annula International Symposium on Computer Architecture*, May 1999.
4. Shubhendu S. Mukherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocol. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.
5. M.F.P. O'Boyle, A.P. Nisbet, and R.W. Ford. A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems. In IEEE Computer Society Press, editor, *Proceedings PACT'96, Boston*, October 1996.
6. Per Stenström, Mats Brorsson and Lars Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993.
7. Jonas Skeppstedt. *Compiler-based approaches to reduce memory access penalties in cache coherent multiprocessors.* PhD thesis, Chalmers University of Technology, April 1997.

8. Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–356, April 1989.
9. S.C. Woo, M. Ohara anf E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.
10. Kattamuri Ekanadham, Beng-Hong Lim, Pratap Pattnaik, and Mark Snir. PRISM: An integrated Architecture for Scalable Shared Memory In *Proceedings of the 4th International Conference on High Performance Computing Architecture*, February 1998.
11. Jean-Thomas Acquaviva and William Jalby : Shared Memory Scientific Applications: a few key properties for optimization scheme In *CEA/DAM or Prism Technical Report*, March 2000.

Hardware Migratable Channels*

David May, Henk Muller, and Shondip Sen

Department of Computer Science, University of Bristol, UK.
http://www.cs.bris.ac.uk/

Abstract. Channels as an essential part of a processor's instruction set were first launched with the Transputer. We have made two major alterations: the semantics of the input and output instruction are changed in order to overlap communication, and channels are allowed to be communicated over channels (higher order communications). All operations can be easily implemented in hardware.

1 Introduction

Communication is at the heart of concurrent systems. It is well known that in order for concurrent systems to work efficiently, we need efficient yet flexible communication primitives. In this paper we focus on a hardware implemented communication primitive (such as found in the Transputer family). We have changed the semantics of the hardware channels in two ways. First, instead of fixed communication channels, we allow channel ends to migrate. Second, we have solved the buffer management issue by requiring the compiler to define where each received message is to be buffered. This paper describes the instructions and protocol, full details including the implementation are in [1].

2 Compiler Directed Input Buffers

The traditional interface for communication over Occam style channels consists of two operations: `input` and `output` [2]. The input operation inputs a number of bytes on a channel on a given address, the output operation outputs a number of bytes on a given channel. In the classic implementation, the output operation blocks until the input operation is executed, whereupon the data is transferred between the two processes and both processes are released. This implements a synchronous transfer in hardware.

Although these semantics are very elegant, and useful for compilers and humans to reason about, they are not the most efficient semantics for implementing channels in hardware. In particular, it is difficult to hide latency. In a naive implementation the sender would first ask the receiver for permission to send the data, after granting permission, the data would be transferred, incurring a quadruple latency.

* This work was partially funded by Hewlett Packard Research Laboratories Bristol, UK

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 545–549, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Solutions solving the latency issue often copy the data; however, we can avoid both excessive latency and copying by redefining the input primitive so that it will store data in a *compiler defined* pre-allocated buffer. (Note that this is different from run time allocated buffers, such as used in Mach [3], or the use of scatter-buffers as used in Solaris [4].) We propose an architecture where each port has an associated memory location where the data is going to be stored, with an input primitive with the following syntax and semantics:

```
input port-register,address-register
```

This primitive will wait for the data to appear on the port and then perform the following actions: swap the address-register and the current buffer location of the port; and send an acknowledgement to the outputting process to signal that the I/O operation has succeeded.

This instruction does not specify where the data of *this* input operation is to be stored, but it specifies where the data of the *next* input operation is to be stored. Typically, the compiler knows where the data will be needed, so it can specify the right memory location. In the worst case, the compiler will have to use two global buffers to store the data alternately. This is no slower than the original copying scheme.

What makes this input operation unique is that it opens the door to implement many compiler optimisations. For example, a loop reading data into an array can be transformed as follows:

```
int a[100], i ;                  int a[100], i ;
channel in_c, out_c ;            channel in_c=&a[0], out_c ;
for( i=0 ; i<100 ; i++ ) {   ⇒ for( i=0 ; i<100 ; i++ ) {
  input( in_c, &a[i], 1 ) ;        inputNEW( in_c, &a[i+1], 1 ) ;
}                                } /* ^^^ Reads into a[i]! */
```

Each port is initialised with an initial memory buffer where the first data item that is going to be read will be stored. Similarly, a loop reading data and processing it can be unrolled once. Note that the `input` operation is still strictly synchronous (unlike, for example, the aioread call in Solaris [4]). We separate synchronisation and data transfer, much like splitting a load instruction in a `prefetch` and `load`. A split output operation (presend + synchronise) is described in [5] and [6].

3 Communicating Ports over Ports

By allowing ports to be sent over ports, we are able to create a communication graph that is no longer fixed. This is accomplished by treating ports as first class objects, similar to the pi-calculus [7]. However, because our channels are synchronous and point to point, they can be moved relatively easily. In a previous study, a relocatable channel end or port was described as an entity of its own [8]; regardless of the medium it was moved over. Although this provided a good primitive to work with, embedding it in a hardware environment proved non trivial. We have now designed a new protocol for moving ports over a network of

virtual channels that is suitable for a hardware implementation, such as the one proposed for MIDAS [9]. This protocol prevents chains of synchronisations and chains of sent data building up. At most two steps are to be performed on any state transition, enabling a trivial implementation using microcode or an FSM.

Each port consists of an entry in a port-table and can be in one of seven states as shown in the state transition diagram, Figure 1(a). The following invariants hold. **EMPTY** No process has a reference to this port. If the state is not empty, then exactly one process has a reference to this port, exactly one other port in the system, c, has this port, t, as its companion port, and the companion port of t is c. **IDLE** No process is performing an input or output on this port at present. **INPUTTING** Exactly one process is performing an input operation on this port. **OUTPUTTING** Exactly one process is performing an output operation on this port. The data has already been pre-sent to the companion port. **BUFFERFULL** Data has been received on this port from the companion port, but no process is performing an input on this port. **MIGRATING** This port is attempting to migrate to another node. No input or output can be performed on this port (for it is being migrated). One other copy of this port may be around on the node where this port was sent to. The companion port will be informed of the migration, when informed, it will resent any pre-send data. **CONFIRMING** The port has been used to read data from. The associated process cannot restart until the inputted port has been completely wired up.

3.1 Protocol

As a simple example, a migrating port is illustrated in Figure 1(b), with three nodes: origin, destination and companion with processes O, D and C on each node respectively. Two channels p and t connect the processes OC and OD. Our objective is to relocate the port p_1 from the environment of process O to D via the transport channel t (the port labels t_i and t_o stand for input and output), the end result being a channel p which connects C to D.

Stage 1: prepare for move When process O attempts to output p_1 over a port, t_0 the channel migration operation will be initiated: **(1a)** The state of port p_1 is set to MIGRATING from either IDLE or BUFFERFULL. This obstructs any other process at the origin from trying to communicate using this port. If the port was previously in the BUFFERFULL state, any buffered messages remain unacknowledged and are retransmitted later to the destination process D. **(1b)** Port t_o, the port over which p_1 is migrating, is stored in the address field of the port-table at the origin. This creates a route for the confirmation message to be sent at a later stage. **(1c)** A data message is sent over t consisting of the companion port of p_1, i.e. the address of p_0. The system then remains in this state until the process where t_i resides commits to input the port p_1. Ownership of t_i may change as it may itself migrate.

Stage 2: input of the port Stage 2 commences when the port transferred over t is inputted by process D on the destination node: **(2a)** A new port is allocated in the port-table. The state of the port is set to IDLE, and the companion-id is

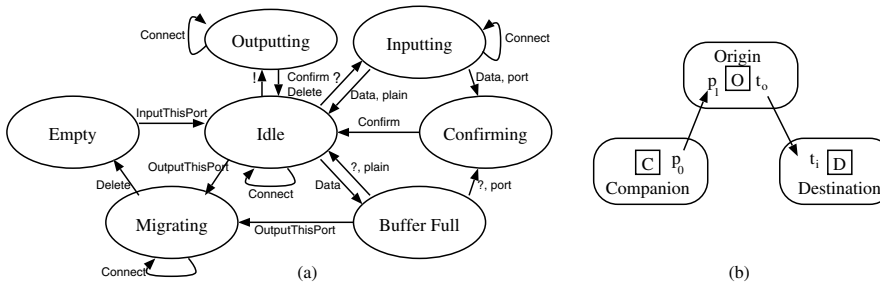


Fig. 1. (a) Complete state machine, (b) Nodes, processes and ports

set to the data read from t_i. The index of the new port is stored on the stack of the process (ready to be used). **(2b)** A message is sent to the companion node requesting port p_0 to be wired up to the newly allocated port. This message consists of the companion-id, and the newly allocated port-index. **(2c)** port t_i is set to CONFIRMING to signal that the port has been read and is being wired up.

Stage 3: wiring up, cleaning up and confirmation The third stage of the protocol starts when the companion node receives the request to wire-up. **(3a)** The companion-id of port p_1 is overwritten with the newly received port-id received from the destination node. This will effectively complete the channel p between C and D. **(3b)** If the state of the companion port was OUTPUTTING, then the unacknowledged message must be re-transmitted (see stage 1 part 1a). If the state is MIGRATING, then both ports of channel p are migrating simultaneously (this situation is discussed in the next paragraph). If the companion is in any state other than MIGRATING then a delete message is sent to the origin node causing deletion of the port-table entry of p_1 and notification for the sending process to proceed. Simultaneously a confirmation message is sent from the origin to the destination node notifying it that the migration process has been completed and that the process owning port t_i may be restarted.

Ports where both ends migrate It is possible that both ends of a port migrate simultaneously. If that is the case, then both ports will always find the companion in a state MIGRATING. The solution is simple: we forward the wiring up message to the new destination node of the companion node, where two cases may arise: **(1)** That node has not yet inputted its data, in this case we simply overwrite the data waiting on the port. **(2)** That node has already inputted its data, and a port has been allocated as a companion node. In that case the newly allocated port has been stored in the process structure of D, so we can update the companion-id of this port. Finally we also send the Delete message out to the origin node, which will cause a confirmation to be forwarded to the destination node, as in stage 3.

Deadlock freedom, progress We do not (yet) have a formal proof that the protocol is deadlock free, but we can intuitively see why it is. As far as network deadlock

is concerned, each node will generate at most one message on acceptance of a message. If the network can always accept a message once a message has been delivered, then all messages will always be delivered. The protocol itself is deadlock free because there is only one situation where the protocol can block: that is in stage 2, when the protocol waits for a process to input on t_i. This only deadlocks if the program transferring ports deadlocks.

4 Conclusions

In this paper we have defined a protocol for transporting data (consisting of either ordinary bits or ports) over channels. The protocol speculatively pre-sends data to the receiver, where it is stored in a buffer. The instruction to input data defines where the next message is to be stored. This allows data transport to overlap with computation, while retaining fully synchronous communication. Using this instruction the compiler can implement a zero-copying protocol without dynamic memory management.

If the input port is moved before the data is actually needed then the data will be resent. The protocol to transport a port from one node to the next performs the same speculative pre-send, but only when the data is actually accepted will the port finally move. This results in a protocol which can be implemented trivially in hardware. Any message coming in will result in a state transition and at most one message being generated. An on-chip implementation will have a fixed number of ports for each processor with an area overhead of around 2Kb of static memory for 256 ports. However, thanks to the fact that ports can migrate, one can always migrate some of the software to another processor if more ports are to be employed.

References

[1] D. May, H. Muller, and S. Sen. Hardware Migratable Channels. Technical Report CSTR-00-005, Department of Computer Science, University of Bristol, March 2000.
[2] INMOS. *The Transputer Databook*, November 1988.
[3] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison Wesley, 1997.
[4] *Solaris Reference Manual.* SUN Microsystems, 1998.
[5] D. Towner and D. May. Optimising Concurrent Software Using Split Communication Transformations. Technical Report CSTR-00-LR (submitted for publication), Department of Computer Science, University of Bristol, Jan. 2000.
[6] M. Goldsmith. The Oxford Occam Transformation system, 1988.
[7] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.
[8] H. L. Muller and D. May. A Simple Protocol to Communicate Channels over Channels. In *EURO-PAR '98 Parallel Processing, LNCS 1470*, pp 591–600, Southampton, UK, September 1998. Springer Verlag.
[9] R. Kirk and A. Hunt. MIDAS–MILAN An Open Distributed Processing System for Audio Signal Processing. *Journal Audio Engineering Society*, 44(3):119–129, Mar. 1996.

Reducing the Replacement Overhead on COMA Protocols for Workstation-Based Architectures

Diego R. Llanos Ferraris, Benjamín Sahelices Fernández, and Agustín De Dios Hernández

Computer Science Department, University of Valladolid, Spain.
{diego, benja, agustin}@infor.uva.es

Abstract In this paper we discuss the behavior of the replacement mechanism of well-known COMA protocols applied to a loosely-coupled multicomputer system sharing a common bus. We also present VSR-COMA, a COMA protocol that uses an advanced replacement mechanism in order to select the destination node of a replacement transaction without introducing penalties due to the increase of the network traffic. Our comparative study of the behavior of different replacement mechanisms in the execution of Splash-2 programs confirm the effectiveness of the VSR-COMA protocol for this kind of systems.

1 Introduction

The use of workstation networks as loosely-coupled multicomputer systems allows to build distributed shared memory architectures with good price/performance ratio [1]. The main drawback of this kind of architectures is the use of a common bus: a slow transmission media that constrains the speedup and the scalability of these systems.

The design of a COMA protocol [4] for workstation-based architectures has already been proposed. COMA-BC [7] is a bus-based COMA protocol that reduces the network traffic using a snoopy-directories hybrid mechanism. This approach leads to a lower number of messages across the network. COMA-BC protocol has been evaluated running different Splash-2 programs [2], and good speedups have been obtained.

COMA-BC, however, presents a major drawback: there is no replacement mechanism. Instead, each attraction memory (AM) has the same size as the shared address space. This approach simplifies the protocol design, because there is no need to replace a block in order to make free space in the local AM, but it does not allow an efficient use of the AM space of each node.

There are two distributed COMA protocols, COMA-F [5] and DICE [3], that incorporate replacement strategies. As we will see, both approaches lead to a considerable overhead when are applied to a common bus-based COMA architecture.

A new bus-based COMA protocol that pretends to solve this problem has been developed. VSR-COMA (Valladolid Smart Replacement COMA) [6] is a

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 550–557, 2000.
© Springer-Verlag Berlin Heidelberg 2000

COMA protocol that incorporates a new replacement mechanism. Every VSR-COMA cache-coherency controller knows the situation of each cache line in each remote AM of the system. This allows to choose the most appropriate destination node without introducing more traffic in the interconnection network. This solution also allows the local cache-coherency controller to make the decision based on more sophisticated, protocol-independent algorithms. In addition, this solution is not affected by the memory pressure.

The rest of this paper is organized as follows: Section 2 discusses in more detail the replacement strategies used in distributed COMA protocols. Section 3 introduces the VSR-COMA protocol. Section 4 describes the replacement strategy used in VSR-COMA. Section 5 makes a speedup comparison based on a simulation study of the protocols mentioned above. Finally, Section 6 shows our conclusions.

2 Replacement Strategies in COMA Protocols

The main problem in the replacement mechanism in distributed COMA protocols is the selection of the destination node. When a node needs to make free space and is the owner of every block in the corresponding set of the local AM, the need of sending the ownership of a block to another node arises. We have two problems. First, which block should be selected. Second, which node should be the destination for the replacement operation. The former is not so difficult to solve: first of all, we will try to transfer the ownership of a block that has another copy in a remote AM. If it is not possible, we need to transfer an "exclusive block", a single-copy block. The latter is more complex, because in a distributed COMA environment the nodes do not know which remote node has enough free space to accept the block.

Two approaches that pretend to solve this problem have been proposed: the random selection of the destination node, used in the COMA-F architecture [5], and the 4-level priority scheme of the DICE protocol [3]. The random selection consists on the selection of the destination node on a random basis. If the node accepts the block, it sends an acknowledgment message: the ownership has been transferred. If not, the remote node sends a negative acknowledgment and the ownership remains in the original node. In this case, the node should choose another node and try again. Note that at high memory pressures, the probability of choosing the "right" node (the one with enough free space) decreases when the number of nodes is increased.

The 4-level priority scheme used in DICE works as follows. The node sends a message asking for the state of the corresponding set of every remote AM. All the remote nodes answer this message with a 4-level code that reflects its situation: i) The node has a copy of the block; ii) the node does not have a copy but it has a free cache line; iii) the node has every block in use, but there is at least one block that could be overwritten (the node does not have the ownership of it) and iv) the node has the ownership of every block in the set. This 4-level code acts as a priority scheme. In the fourth case, the protocol establishes

that the replaced block must be exchanged with the required block -that is, a swapping technique. DICE replacement technique allows the node that starts the replacement to choose the best destination node, but at a high cost: for n nodes, $n + 2$ messages are needed to complete the replacement transaction (one request, $n - 1$ responses, the ownership transfer and its acknowledgment).

As we can see, both systems require to send several messages to complete the replacement transaction. In addition, the number of required messages increases with the number of nodes in both systems. This leads to a considerable overhead in COMA systems that runs over networks of workstations, where the bus speed is the main bottleneck.

3 The VSR-COMA Protocol

The design goals of VSR-COMA are focused on the construction of a COMA protocol useful for a common bus multicomputer system. The main goal is to reduce the network traffic, based on the broadcast feature of the common bus. Since each message sent by a processing node is received by every node in the system, each node could theoretically know the block distribution in the remote AMs. This characteristic will allow the VSR-COMA cache-coherency controllers to trace the situation in all the remote AMs, and therefore to choose the best destination node for a replacement request without introducing extra messages.

The cache-coherency controller of a VSR-COMA node manages three basic data structures: the directory table, the state+tag table, and the replacement table. The directory table holds the information related to the owner of each block in the system. Every request in VSR-COMA protocol should be sent to the owner of the block. The state+tag table keeps record of the state of every cache line in the local AM, with the corresponding tags. Both structures are similar to those found in other directory-based COMA architectures.

The third data structure is the replacement table. The replacement table keeps track of the state of every cache line in the *remote* AMs, with the corresponding tags. This information is updated by the cache controller as follows: when the cache controller receives an event (remember that in a common bus architecture, every node sees every event generated in the system), it updates its directory and replacement tables with the information present in this event, i.e. if the event is a read request, every cache controller notices that the sender wants to read a block, and therefore it updates the corresponding data entry in the replacement table. To do this, each event carries on the tag information of the block requested and also the number of the cache line *inside the set* that will be updated. This information is enough for the cache-coherency controllers to keep track of the evolution of all the remote AMs.

This approach has a possible drawback. It would seem that we would have a considerable memory overhead in order to maintain such information replicated in every cache controller. This problem is not so significative: our results shows a memory overhead two to three times higher for our system than for a similar

system without replacement information, and does not exceed a 10% overhead [6].

3.1 Events, States, and Operations

There are two types of states in VSR-COMA: *stable* states and *transient* states. The stable states are similar to the states we found in DICE [3]: Inv (the cache block is not valid), Shared (the cache block is valid for reading, that is, the local node can read the information but it cannot overwrite it, since the local node is not the owner of the block), SharOwn (the local node is the owner of the block but possibly there are other copies of the block in remote AMs) and Excl (the local node is the owner of the block and there is exactly one copy of it).

Transactions can be overlapped in a single bus network. The utilization of this kind of bus (called "split-transaction bus") leads to the need of transient states in the protocol, indicating that a particular operation is still in progress. VSR-COMA uses three transient states: InvWExcl ("invalid, but waiting for an Exclusive block"), InvWExcl ("invalid, but waiting for a Shared copy of a block"), and WExport ("valid, but exists an export transaction in progress").

In VSR-COMA terminology, every message sent between the nodes is called an *event*. A pair of request-response events is called a *transaction*. Every event in VSR-COMA protocol has a source and a destination node. There are nine different events:

BusRreq : Bus read request.
BusRack : Bus read acknowledgment.
BusWreq : Bus write request.
BusWack : Bus write acknowledgment.
BusFInv : Bus fast invalidation. It is used by a node that has a SharOwn block and wants to modify it. This event invalidates every copy of this block in the remote AMs.
BusEXreq : Bus export request. This event includes the exported block, and is used to send the ownership of a block to another node.
BusEXack : Bus export acknowledgment.
BusEXnak : Bus export negative acknowledgment. The destination node cannot accept the incoming block due to space problems. The first node keeps the ownership of the block.
BusRACE : Bus race condition. This event is used when a node receives a request about a block that has another owner. This situation can be produced due to a transaction overlapping.

Finally, VSR-COMA establishes a simple memory protocol that allows the processors to request operations to the cache controller of each local node. This protocol has the following *memory operations*: PrRd (processor read), PrWr (processor write), PrTAS (processor test-and-set) and PrFAI (processor fetch-and-increment). The last two operations are used to implement synchronization operations: semaphores and barriers.

3.2 State Transition Diagram

Figure 1 shows the VSR-COMA state transition diagram. Note that in the figure we only consider the PrRd and PrWr memory operations, because PrTAS and PrFAI memory operations behave exactly like a PrWr operation from the network protocol point of view.

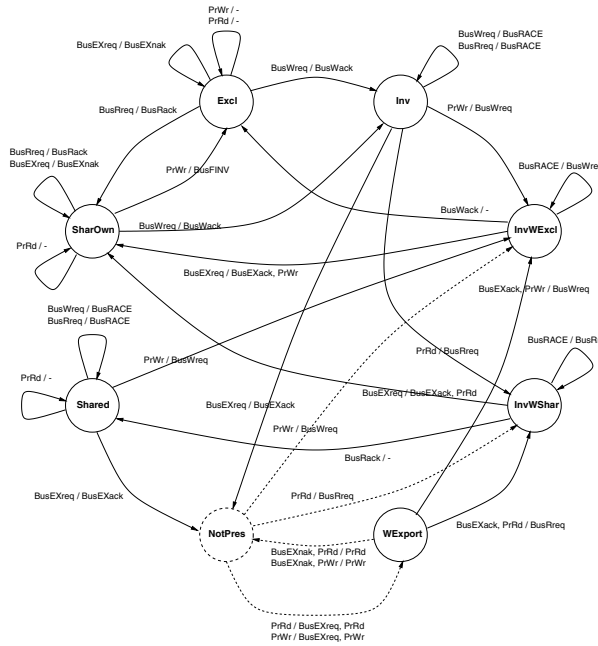


Fig.1. VSR-COMA state transition diagram. The transitions are labeled as Operation-or-Event received / Event generated.

The diagram of fig. 1 has eight different states. The eighth state is Not Present. This is not a block state. Instead, this state reflects the possibility of that the block is currently not present in the system. In this case we need to know the situation of the rest of the cache lines in our set, in order to decide what to do. If the block is not present and we have at least one block in Inv or Shared state, this block will be overwritten, changing its state to InvWShar or InvWExcl state. If the block is not present and every block in our set has the Excl or SharOwn states, we need to perform a replacement operation. Note that the selection of the destination node of the exported block does not depend on the protocol. This characteristic allows the designer to explore different replacement strategies without modifying the protocol, leading to a more flexible behavior. In the following section we will examine the replacement strategy currently used in VSR-COMA.

4 VSR-COMA Replacement Strategy

As was explained in section 3, VSR-COMA allows each cache-coherency controller to know the situation of every remote AM in the system. Note that this information is not complete, because new events can be produced at the same time the controller is checking this information, due to the race conditions inherent to the bus architecture. This information, however, can be effectively used to select the destination node for an export operation.

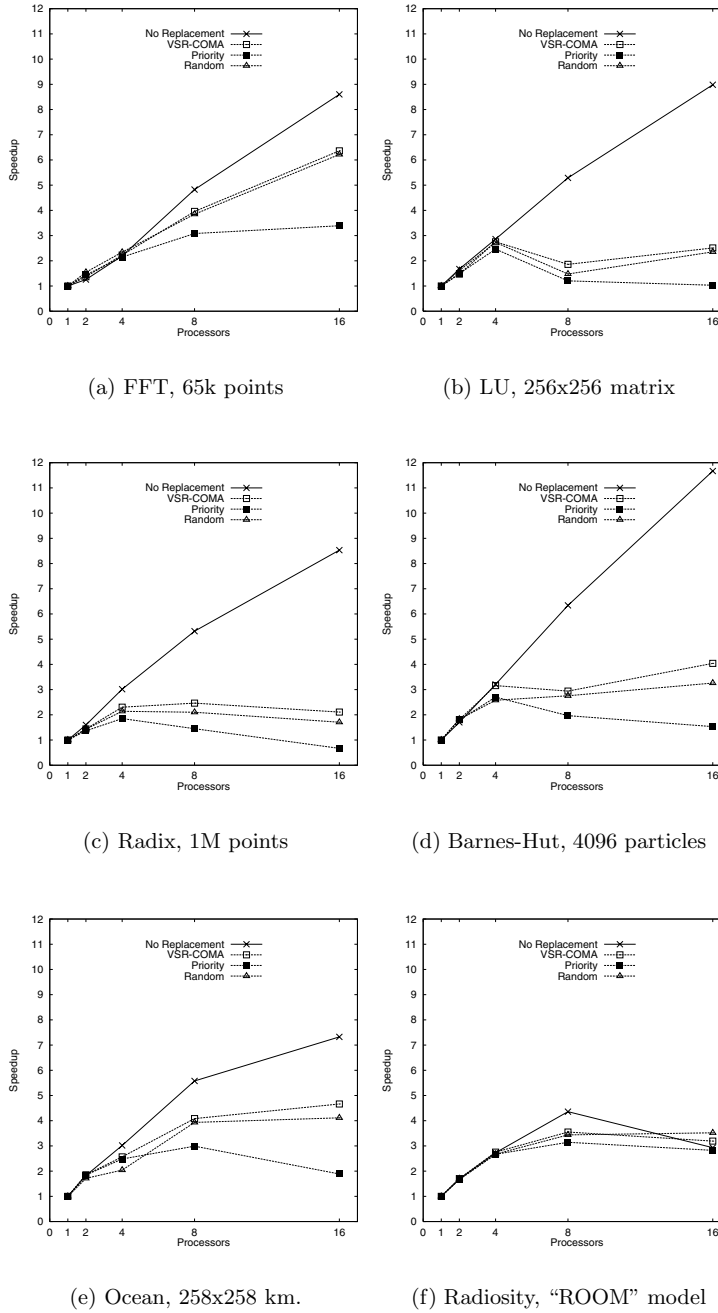
VSR-COMA uses the following selection algorithm:

1. If the block we want to export is in SharOwn state, a node with a Shared copy of this block is chosen.
2. Otherwise, we look for a node with the block we want to export in InvWExcl state (that node is currently requesting our block to the wrong owner).
3. Otherwise, we look for a node with the block we want to export in InvWShar state (that node is currently requesting a copy of our block to the wrong owner).
4. Otherwise, we look for a node with the block we want to export in Inv state (that node has been using our block in the recent past).
5. Otherwise, we look for a node with *any* block of the set in Inv state.
6. Otherwise, we look for a node with *any* block of the set in Shared state. This selection will force the destination node to discard a Shared block.
7. At this point, it seems that every block in the corresponding set of the remote AMs is in Excl, SharOwn or a transient state. This situation is possible at high memory pressures. The solution is to look for a node with any block in InvWShar state: when the replacement request arrives to that node, it is possible that the node has already completed its BusRreq and has a Shared copy that could be overwritten. If not, the node will respond with a BusEXnak event and the local node will start the node selection process from the beginning.
8. Otherwise, we look for a node with a block in InvWExcl state. The replacement request will probably be denied, but at this point there is no alternative. However, this is an extremely infrequent and transient situation: after this attempt the situation will surely change.

We would have more than one node that meet the requirements in each step. In this case, the node with less blocks in ownership in the set is chosen. This selection method leads to a better balance of the ownerships between the AMs.

5 Results

Figure 2 shows the speedup comparison using six well-known programs of the Splash-2 benchmark suite [2]. The simulation results have been obtained considering a set of RISC workstations at 167 MHz, and a Myrinet-type interconnection network. We have considered 4-way associative AMs, 256 bytes block size and



(a) FFT, 65k points

(b) LU, 256x256 matrix

(c) Radix, 1M points

(d) Barnes-Hut, 4096 particles

(e) Ocean, 258x258 km.

(f) Radiosity, "ROOM" model

Fig.2. Speedup comparison for the three replacement strategies studied above with a no-replacement approach.

an 80 percent memory pressure. At this high memory pressure there are many remote misses, and the overhead due to replacement operations increases.

Figure 2 shows the behavior of VSR-COMA protocol using four different replacement algorithms: i) the random node selection used in COMA-F; ii) the four-level priority scheme used in DICE; iii) the selection algorithm for VSR-COMA described in section 4, and iv) the no-replacement approach proposed by COMA-BC. In this sense, COMA-BC acts as an infinite-way associative system, and therefore there is always enough space to avoid a replacement. The speedup results are heavy influenced by the use of a loosely-coupled, workstation-based architecture, but we can see that the VSR-COMA destination node selection algorithm works better than the rest of replacement algorithms at high pressures, providing a speedup over random selection that reaches 124% in Radix and a speedup over priority-based mechanisms that reaches 315%, again in Radix.

6 Conclusions

Our results confirm that VSR-COMA is a valid alternative to build COMA machines with a network of workstations that share a common bus. We have also proposed a replacement algorithm that leads to better results than other well-known replacement algorithms for this kind of systems.

The design of the VSR-COMA protocol allows the designer to explore different replacement algorithms without modifying the protocol, leading to a more flexible behavior.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, pages 54–64, February 1995.

[2] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and metodological considerations. In *Proceedings of the 22nd Anual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[3] Sangyeun Cho, Jinseok Kong, and Gyungho Lee. Coherence and Replacement Protocol of DICE - A Bus Based COMA Multiproccesor. *Journal of Parallel and Distributed Computing*, pages 14–32, April 1999.

[4] Fredrik Dahlgren and Josep Torrellas. Cache-only memory architectures. *IEEE Computer*, pages 72–79, June 1999.

[5] Truman Joe. *COMA-F: A Non-hierarchical Cache Only Memory Architecture*. PhD thesis, Department of Electrical Engineering, Stanford University, 1995.

[6] Diego R. Llanos Ferraris. *VSR-COMA: Un protocolo de coherencia cache con reemplazo para sistemas multicomputadores con gestión de memoria de tipo COMA*. PhD thesis, Departamento de Informática, Universidad de Valladolid, España, Abril 2000.

[7] Benjamín Sahelices Fernández, Juan Illescas, and Luis Alonso Romero. COMA-BC: A cache only memory architecture multicomputer for non-hierarchical common bus networks. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 502–508, 1998.

Cache Injection: A Novel Technique for Tolerating Memory Latency in Bus-Based SMPs

Aleksandar Milenkovic, Veljko Milutinovic

School of Electrical Engineering, University of Belgrade,
P. Box 35-54, 11120 Belgrade, Yugoslavia
{emilenka, vm}@etf.bg.ac.yu

Abstract. Cache misses and bus traffic are key obstacles to achieving high performance of bus-based shared memory multiprocessors using invalidation-based snooping caches. To overcome these problems, software-controlled techniques for tolerating memory latency can be used, such as cache prefetching and data forwarding. However, some previous studies have shown that cache prefetching is not so effective in bus-based shared memory multiprocessors, while data forwarding is not easy to implement in this environment. In this paper, we propose a novel technique called cache injection, which combines consumer and producer initiated approaches, as well as the broadcasting nature of bus. Performance evaluation based on program-driven simulation and a set of eight parallel benchmark programs shows that cache injection is highly effective in reducing coherence misses and bus traffic.

1 Introduction

Private caches are essential to reduce the bus traffic and the memory latency in bus-based shared memory multiprocessors (SMPs). In such systems, snooping write-invalidate cache coherence protocols are commonly accepted as an effective approach to keep the data coherent [1]. However, the problem of high memory latency is still the most critical performance issue in these systems. One way to cope with this problem is to tolerate high memory latency by overlapping memory accesses with computation. The importance of techniques for tolerating high memory latency in multiprocessor systems increases, due to the widening speed gap between CPU and memory, high contention on the bus, bus traffic caused by data sharing between processors, and the increasing physical distances between processors and memory.

Software-controlled cache prefetching is a widely accepted consumer-initiated technique for tolerating memory latency in multiprocessors, as well as in uniprocessors. In software-controlled cache prefetching, a CPU executes a special `prefetch` instruction that moves a data block (expected to be used by that CPU) into its cache, before it is actually needed [2]. In the best case, the data block arrives at the cache before it is needed, and the CPU load instruction results in a hit. However, for many programs and sharing patterns (e.g., producer-consumer), producer-initiated data transfers are a natural style of communication. Producer initiated primitives are known

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 558-566, 2000.
© Springer-Verlag Berlin Heidelberg 2000

as data forwarding, delivery, remote writes, and software-controlled updates. With data forwarding, when a CPU produces the data, in addition to updating its cache, it sends a copy of the data to the caches of the processors that are identified by compiler or programmer as its future consumers [3]. Therefore, when consumer processors access the data block, they find it in their caches.

Most of the studies [2-8] examined the effectiveness of cache prefetching and data forwarding in CC-(N)UMA architectures, except [9], which examined the potential of cache prefetching in bus-based SMPs. This study reported poor effectiveness of cache prefetching, despite the assumed high memory latency. The main reasons for that are the following. First, prefetching increases bus traffic. Since bus-based architecture is very sensitive to changes in bus traffic, it can result in performance degradation. Second, too early initiated prefetching can negatively affect data sharing. Last, current prefetching algorithms are not so effective in predicting coherence misses. Actually, coherence misses represent the biggest challenge for designers, especially as caches become larger and they dominate the performance of parallel programs.

On the other side, complexity of implementation and compiler algorithm restricts applicability of data forwarding in bus-based architectures. Dahlgren et al. explored the effectiveness of the software-controlled update in bus-based SMPs, where a special instruction initiates an update of all invalid copies of the specified cache block in the system [10]. This approach requires less sophisticated compiler support since it does not require identification of future consumers, and it can be implemented at low cost. However, it is less flexible than classic data forwarding as defined in [2], because it does not allow forwarding to the processors not having the invalid copies of the data block. In paper [11], Anderson and Baer showed that the technique called read snarfing could be very effective in reducing the number of coherence misses and the bus traffic in bus-based SMPs. With read snarfing, a data block that is transferred on the bus as a read response not only updates the node that requested it, but also updates all other caches having the block invalidated. Read snarfing is a hardware-based technique, easy to implement. However, it is based on the heuristic that all blocks that are invalid will be needed in the future, and its effectiveness highly depends on cache size. In the system with relatively small cache size, the invalid cache blocks will be probably displaced from the cache, so read snarfing is not applicable.

In this paper, we propose a novel software-controlled technique called cache injection, aimed to reduce coherence misses and bus traffic. Using advantages of the existing techniques and the characteristics of bus-based architectures, cache injection overcomes some of the shortcomings of the existing techniques, such as: (a) bus and memory contention, (b) negative impact on data sharing and instruction overhead in the case of cache prefetching, and (c) compiler and implementation complexity in case of data forwarding. The proposed technique can be combined with the existing ones in order to raise performance in bus-based SMPs.

In the following section, we define cache injection and discuss its implementation in a bus-based shared memory multiprocessor. Section 3 describes experimental methodology. Section 4 presents results of the experiments. Section 5 summarizes current and discusses the possible future work.

2 Cache Injection

In cache injection, a consumer predicts its future needs for shared data by executing an `OpenWin` instruction. This instruction only stores the first and the last address of successive cache blocks, in a special local injection table. This address scope is called address window. There are two main scenarios when cache injection could happen: during the read bus transaction (injection on first read) or during the software-initiated write-back bus transaction (injection on write-back).

Injection on first read is applicable when there is more than one consumer. Each consumer initializes its injection table according to its future needs. When the first one among consumers executes a load instruction, it sees cache miss and initiates a bus read transaction. During this transaction, each cache controller snoops the bus and if there is an injection hit, the processor stores the block into its cache (Fig. 1a). Hence, in case of multiple consumers, only one read bus transaction is needed to update all consumers, if they all have initialized their injection tables.

Injection on write-back bus transaction is applicable when shared data exhibit both 1-Producer-1-Consumer (1P-1C) and 1-Producer-Multiple-Consumers (1P-MC) patterns. In these scenarios, each consumer also initializes its injection table. At the producer side, after the data producing is finished, the producer initiates write-back bus transactions in order to update the memory, by executing an `Update` instruction. During this transaction, all consumers snoop the bus, and if they find injection hit, they catch the data block from the data bus and store it into their caches (Fig. 1b).

The above definition of cache injection assumes a bus-based SMP where each processor has one or more levels of cache memory and a write-back invalidate cache-coherence protocol based on snooping. Hardware support for cache injection includes injection table, proposed instructions (Fig 1c), and a negligible modification of the bus control unit. The injection table is implemented as a part of the cache controller. Each entry includes two address fields, *Laddr* and *Haddr*, which define the first and the last address of an address window, respectively, and a valid bit *V*. We use the random replacement policy.

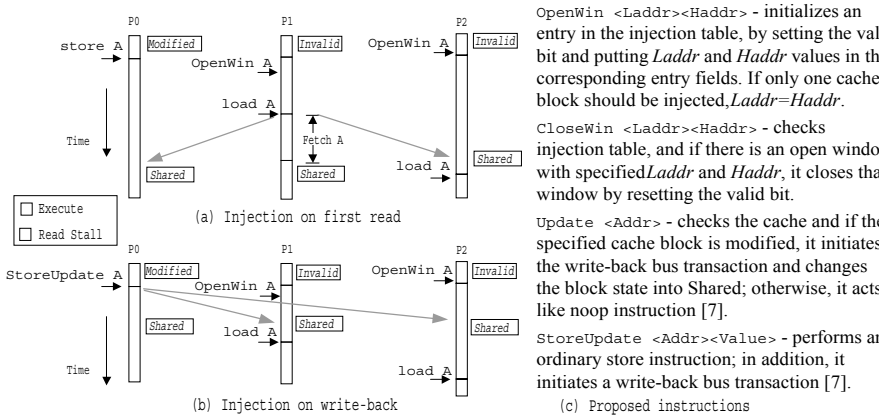


Fig. 1. Cache injection mechanism.

3 Experimental Methodology

We evaluate the performance impact of cache injection using Limes [12] – a tool for program-driven simulation of SMPs. A synchronization kernel (LTEST), three parallel test applications well suited to demonstrate various data sharing patterns (PC, MM, Jacobi), and four applications from SPLASH-2 suite (Radix, FFT, LU, Ocean) [13] are used in the evaluation. They are all written in C using the ANL macros to express parallelism and compiled by gcc with the optimization flag –O2. Proposed instructions for cache injection support are hand-inserted into the applications. For each application we compare the number of read misses and the bus traffic for the base system (B), the system with read snarfing (S), the system with software-controlled update and read snarfing (U), and the system with cache injection (I).

The modeled architecture is bus-based SMP containing 16 processors with the MESI write-back invalidate cache coherence protocol. The bus supports split transactions and uses round robin arbitration scheme. We assume a single-issue, in-order processor model with blocking reads. Processors execute a single cycle per instruction. Each processor includes only first level cache memory. We assume that instructions always hit into the cache. Cache hit is solved without penalty. The relevant system parameters are the following: cache line size is 32B, data bus width is 8B, snoop cycle is 2pclk (pclk - processor cycle), and write-back buffer size is 32B. The read and the read-exclusive bus transactions include the request and the response phases. The memory read cycle defines time needed to retrieve a requested block from memory; assumed value is 20pclk. A two-word transfer via the data bus takes 2pclk; hence, the block transfer takes 8pclk. It is assumed that the memory controller buffer has enough capacity to accept each block during write-back bus transactions at the data bus speed. A 128-entry injection table was used in the evaluation.

We have used the following data sets: 1000 acquire requests per processor for LTEST, 128×128 shared matrix and 20 iterations for PC, 128×128 matrix for MM, 256×256 matrix and 20 iterations for Jacobi, 128K keys with 8-bit digit for Radix, 256×256 matrix with 8×8 blocks, 256×256 for FFT, and 130×130 for Ocean.

The aim of our evaluation is to first determine the upper bound of performance benefit of cache injection, before we start developing compiler support. Hence, we use simple heuristics based on application behavior to insert instructions for cache injection by hand. Support for injection of synchronization variables is accomplished using injection on first read, since this approach does not require any modification of synchronization operations. This support is quite simple and includes the initialization of the injection table before a synchronization event and the invalidation of the corresponding entry in the injection table after the synchronization is finished. It is clear that inserting instructions to support injection of synchronization variables can be solved by using macros that expand synchronization operations. Hence, the true challenge is the compiler support for injection of true-shared data. If there is a 1P-MC sharing pattern, injection on first read or injection on write-back can be used. Although injection on write-back may be more efficient, we use injection on first read because it implies no action at the producer side. However, if sharing pattern is 1P-1C, we have to use injection on write-back.

4 Results

For synchronization kernel LTEST both read snarfing and cache injection are highly effective: read snarfing reduces the number of read misses and the bus traffic for 90% and 88%, respectively, while cache injection for 92% and 90%. Since the effectiveness of these two techniques is approximately the same for synchronization operation, we do not model synchronization requests on the bus in the experiments with the parallel applications. In this way, we avoid the over-estimation of the synchronization overhead due to relatively small data sets.

Fig. 2 shows the number of read misses and the bus traffic for parallel applications, normalized to the base system, when the caches are relatively small (left) and relatively large (right). For all applications cache injection (I) outperforms read snarfing (S) and software-controlled update with read snarfing (U). The effectiveness of solution I relative to solutions S and U is higher in the system with small caches: invalid blocks are frequently displaced from the cache and in that case snarfing is not applicable. Next, cache injection can be effective in reducing cold misses, when there are multiple consumers of shared data, while snarfing can eliminate only coherence misses. Last, cache injection increases the possibility of successful injection, since the time window during which a block can be injected is software-controlled. The rest of this section explains the data sharing patterns and injection support, and discusses results for each application.

PC. In PC, the coherence misses dominate since each processor modifies its assigned sub matrix, which is read by all other processors in the next iteration (1P-MC sharing pattern). Solutions S and U are almost as effective as cache injection, in the system with large caches. Slight advantage of cache injection is due to elimination some of cold misses. However, in the system with small caches, solutions S and U are not effective at all. The main reason for this is that invalidated data, which should be updated during the next bus read or write-back transaction, are displaced from the cache due to cache conflicts.

MM. MM is a parallel version of matrix multiplication A=AxB, where each processor computes elements of the assigned sub matrix of matrix A. As all processors only read elements of the shared matrix B, to support cache injection each processor defines an address window encompassing the whole matrix B. Cache injection reduces the number of read misses and bus traffic for 92% and 88%, respectively, in the system with small caches, and for 91% and 77% in the system with large caches. Here solutions S and U are not effective at all since the shared data are read only predominantly. The efficiency of cache injection does not increase as the cache size increases. The system with small caches exploits the benefit of multiple injections of data which are thrown out of the cache due to cache conflicts, while in the system with large caches the elements of matrix B are injected only once during the execution.

Jacobi. Jacobi is a method for solving partial differential equations and iterates over a two-dimensional array. In each iteration, every matrix element is updated to the average of its four neighbors. All processors are assigned roughly equal chunks of rows. Neighboring processors share the rows on a chunk's boundary, so there is a predominantly 1P-1C sharing pattern. Consequently, we have to apply the injection on write-back. Solution S is not effective at all, while solutions U and I are equally effec-

tive and reduce the number of read misses for 47% in the system with large caches; in the system with small caches solution I is slightly more effective.

Radix. Radix sorts integer keys using the three-phase iterative radix-sorting method. The injection of the global histogram *rank* is applied in the first phase of iteration. Each processor initializes the injection table to accept the elements of *rank* array currently being updated by the next processor, which should insert an `Update` instruction after the last write in the cache block. In the second phase, each processor computes its *rank_ff*, using the global histogram *rank* and local histograms *rank_me* of all processors with lower ID. As there are multiple consumers, we use the injection on first read. In the last phase, there is an irregular all-to-all communication, so we did not use the injection in this phase. In the system with small caches, solutions S, U, and I reduce the number of read misses for 7%, 8%, and 21%, and the bus traffic for 4%, 3%, and 9%, respectively. In the system with large caches, they reduce the number of read misses for 18%, 21%, and 26%, and the bus traffic for 10%, 10%, and 12%, respectively.

FFT. FFT executes the 1-D version of the six-step FFT algorithm. The data set consists of the *n* complex data points to be transformed, and *n* complex data points referred as the *roots of unity*, both organized as $\sqrt{n} \times \sqrt{n}$ matrices, which are partitioned among processors in contiguous chunks of rows. In the algorithm steps 2, 3, and 5, each processor modifies only its assigned chunk of rows. In the steps 1, 4, and 6, the matrix is transposed: the processor communication is all-to-all, and the data-sharing pattern is 1P-1C. A producer inserts `Update` instructions before the transposing step, while a consumer initializes the injection table to inject the corresponding data. Solution S is not effective since there is predominantly 1P-1C sharing pattern. In the system with small caches, the effectiveness of solutions U and I is limited by conflicts in caches; the number of read misses is reduced for 3% and 8%, respectively, while the bus traffic is increased for 11%, and 7%, respectively. In the system with large caches, solution I is highly effective and reduces 46% of read misses, and 12% of bus traffic, while solution U reduces 30% of read misses and 1% of bus traffic.

LU. LU factors a dense matrix into the product of lower triangular and upper triangular matrices. The matrix is divided into blocks; a block ownership is assigned using 2D-scatter decomposition, with blocks being updated by the processor that owns them. Outer loop iterates over the diagonal blocks. In the second phase of the iteration *k*, the processors that own the perimeter blocks update those blocks, using the diagonal block A_{KK}, modified in the previous phase. As there are more consumers, each processor inserts instructions to support the injection of the diagonal block. In the third phase, the processors modify the interior blocks, using the corresponding perimeter blocks. In this phase, there are also more consumers, so at the beginning of the phase each processor inserts the instructions to support the injection of the corresponding perimeter blocks. Solution I outperforms solutions S and U; it reduces the number of read misses and bus traffic for 30% and 22%, respectively, in the system with small caches, and for 38% and 31% in the system with large caches.

Ocean. Ocean simulates large-scale ocean movements. Data set consists of the uniform two-dimensional grids with $n \times n$ non-border points, partitioned among processors in square-like sub grids. Most of time, the application solves partial differential equations using the red-black Gauss-Seidell equation solver. The injection of true-shared data is implemented predominantly in the phase of the solving of partial differential equations. Generally, a processor communicates with four neighbor processors (Top, Bottom, Left, Right); the data-sharing pattern is 1P-1C. A producer initiates update of the consumer cache with data to be used in the next iteration. A consumer initializes the injection table to accept the last row of the sub grid assigned to the processor Top, first row of the Bottom, left column of the Right and right column of the Left. In the system with small caches, solutions S, U, and I reduce the number of read misses for 14%, 17%, and 25%, and the bus traffic for 19%, 16%, and 28%, respectively. In the system with large caches, solutions S, U, and I reduce the number of read misses for 28%, 35%, and 48%, and the bus traffic for 34%, 30%, and 44%, respectively.

Additional experiments not presented in this paper, which varied architectural parameters, show that the efficiency of cache injection increases with the number of processors in the system, cache memory size, and memory read cycle time. When the number of processors increases, the percentage of shared data increases, as well as the number of sharers, hence the benefit of injection increases due to lowering the overall miss rate and reducing the bus traffic. Larger caches reduce probability of collision of the injected data and the current working set. If the memory read cycle time is longer, there is more to gain by reducing the read stall time.

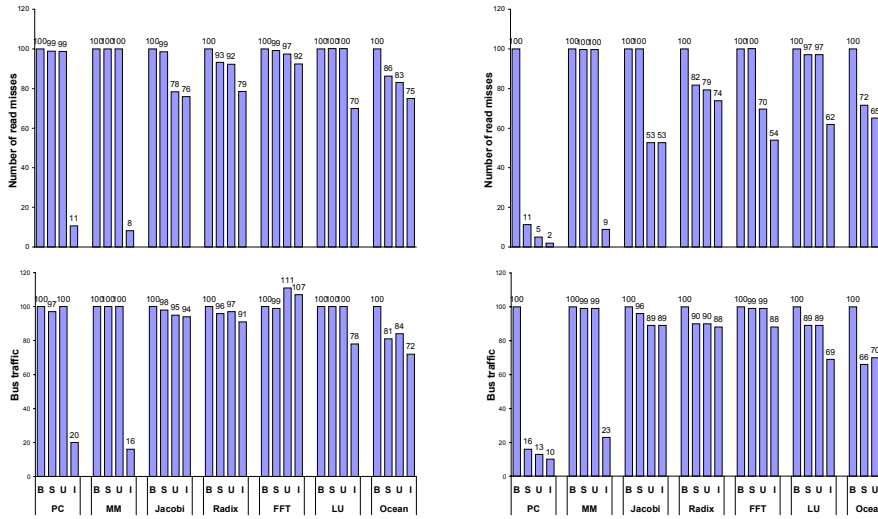


Fig. 2. Number of read misses (upper) and bus traffic (lower) relative to the base system. Cache_size=64/128KB (128KB for FFT, LU, Ocean) (left), and Cache_size=1024KB (right).

5 Conclusion

This paper presents a novel software-controlled technique for tolerating memory latency in bus-based SMPs. This technique, called cache injection, has been developed in order to overcome some of the shortcomings of the existing techniques, cache prefetching, software-controlled update, and read snarfing, combining advantages of these techniques and inherent characteristics of bus-based architectures.

Experimental analysis, based on execution driven simulation, showed highly effectiveness of cache injection in reduction of the number of read misses and the bus traffic, compared to the base system. In addition, it provides further improvements compared to the systems with read snarfing and software-controlled update.

Possible future research includes developing and implementation of a compiler algorithm for inserting instructions to support injection of shared data. Another direction is to implement some kind of cache injection in scalable cache coherent shared memory multiprocessors.

References

1. Culler D., Singh J. P., Gupta A.: Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, San Francisco, CA (1998)
2. Mowry T.: Tolerating Latency Through Software-Controlled Data Prefetching. Ph. D. Thesis, Stanford University, (1994)
3. Koufaty D. A., Chen X., Poulsen D. K., Torrellas J.: Data Forwarding in Scaleable Shared Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Technology, Vol. 7, No. 12. (1996) 1250-1264
4. Byrd, G. T., Flynn M. J.: Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. Proceedings of the IEEE, vol. 87, no. 3. (1999) 456-466
5. Ramachandran U., Shah G., Sivasubramaniam A., Singla A., Yanasak I.: Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. Proceedings of the Supercomputing'95, vol. 2. (1995), 1737-1775
6. Shafi H. A., Hall J., Adve S., Adve V.: An Evaluation of Fine-Grain Producer Initiated Communication in Cache-Coherent Multiprocessors. Proceedings of the 3rd HPCA. (1997) 204-215
7. Skeppstedt J., Stenstrom P.: A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. Proceedings of the PACT'95, IEEE Computer Society Press. (1995) 69-78
8. Trancoso P., Torrellas J.: The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding. Proceeding of the 25th ICPP, IEEE Computer Society Press, Vol. 3. (1996) 79-86
9. Tullsen D., Eggers S.: Effective cache prefetching on bus-based multiprocessors. ACM Transactions on Computer Systems, Vol. 13, No. 1. (1995) 57-88
10. Dahlgren, F., Skeppstedt, J., Stenstrom, P.: Effectiveness of Hardware-Based and Compiler-Controlled Snooping Cache Protocol Extensions. Proceedings of the HiPC. (1995) 87-92

11. Anderson, C., Baer, J.-L.: Two Techniques for Improving Performance on Bus-Based Multiprocessors. Proceedings of the 1st HPCA. (1995) 256-275
12. Magdic, D.: Limes: A Multiprocessor Simulation Environment. TCCA Newsletter, March 1997. 68-71
13. Woo S. C., Ohara M., Torrie E., Singh J. P., Gupta A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. Proceedings of the 22nd ISCA, (1995) 24-36

Adaptive Proxies: Handling Widely-Shared Data in Shared-Memory Multiprocessors

Sarah A.M. Talbot and Paul H.J. Kelly

Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom

Abstract. A performance bottleneck arises in distributed shared-memory multiprocessors when there are many simultaneous requests for the same data. One architectural solution is to distribute read requests to nodes other than the home node: these other nodes act as intermediaries (*i.e.* proxies) in obtaining the data, and combine requests for the same data. Adaptive proxies use proxying during the proxying period, which varies depending on the level of run-time congestion. Simulation results show that adaptive proxies give performance improvements for all our benchmark applications.

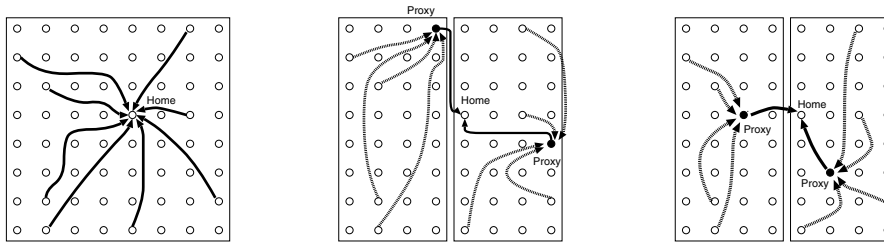
1 Introduction

In a cache-coherent non-uniform access (cc-NUMA) shared-memory multiprocessor, remote access to each processor's memory and local cache is managed by a "node controller". In large configurations, unfortunate ownership migration or home allocations can lead to the concentration of requests for data at particular nodes. This results in the performance being limited by the service rate or "occupancy" of an individual node controller [3].

In this paper we present an adaptive proxy cache coherency protocol, which alleviates contention for widely-shared data, and can do so without adversely affecting any of the applications we have simulated. The adaptive proxy scheme requires no modification or annotation to the application code. The additional protocol complexity and hardware requirements are small: proxying could probably be added to a typical firmware node controller with no hardware change. In our earlier work on proxies, any data obtained by a node acting as a proxy was cached in the processor's second level cache [7]. This was done deliberately to increase the combining effect, *i.e.* further read requests for that data can be satisfied at the proxy. However, the drawbacks include increased sharing list length, cache pollution, and delays to the local processor and node controller processing. The results in this paper include two new caching options: not caching proxy data, and using a separate buffer for proxy data (with access latencies the same as for accessing DRAM).

The rest of the paper is structured as follows: Section 2 introduces adaptive proxies. Our simulated architecture and experimental design are outlined in Section 3. The results of execution-driven simulations for a set of eight benchmark

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 567–572, 2000.
© Springer-Verlag Berlin Heidelberg 2000



(a) Without proxies (b) With two proxy clusters (read data line l) (c) Read next data line (read data line $l+1$)

Fig. 1. Contention is reduced by routing reads via a proxy

programs are presented in Section 4. Finally, in Section 5, we summarise our conclusions and give pointers to further work.

2 Adaptive Proxies

In the proxy scheme, a processor issuing a read request for remote data sends the request message to another node, which is known to act as a proxy for that data line, rather than going directly to the data line's home node [7]. The number of proxy clusters ($\mathcal{NPC}$) is 2 in the example shown in Fig. 1, *i.e.* each processing node has been allocated to one of two sets (this can be done on the basis of network locality). Home node congestion is the run-time trigger for using proxies. In large-scale systems it is impractical to provide enough buffering at each node to hold all the incoming messages, and a commonly adopted strategy handles a `read-request` that reaches a full buffer by sending a negative acknowledgement (a NAK) back to the requester. The results for reactive proxies were encouraging [7], but the scheme suffered from incurring the delay (before the NAK arrives to signal that a proxy read request is needed) each time data is required from a congested home node.

Adaptive proxies use the arrival of a NAK'd `read-request` message to trigger the start of a proxy-period, *i.e.* a time during which any further `read-request` messages destined for the home node are replaced with `proxy-read-request` messages. The proxy-period is modified according to the level of NAKs, using a random walk policy [1]. The probability of a NAK (from a particular home node) occurring within an upper time limit of the last NAK (from that home) is high if the last "inter-NAK" period was less than the upper time limit.

The adaptive proxy policy is controlled by the following parameters: current time T_{curr}, PP_{unit} is one unit of proxy-period time (set to 1000 cycles), PP_{max} is the maximum proxy-period (set to 50), PP_{min} is the minimum proxy-period (set to 1). Each node controller x is extended with two vectors: $LB_{(x,y)}$ gives for each remote node y the time at which the last (NAK) message was received at client node x from node y, and $PP_{(x,y)}$ maintains the current proxy-period for

reads by this client x to each remote node y. The arrival at client node x of a NAK from home node y will trigger the adjustment of $PP_{(x,y)}$ as follows:

$$PP_{(x,y)} = min(PP_{max}, PP_{(x,y)} + 1) \textbf{ if } (T_{curr} - LB_{(x,y)}) < (PP_{unit} \times PP_{max})$$
$$max(PP_{min}, PP_{(x,y)} - 1) \textbf{ otherwise}$$

The choice of suitable values for PP_{max}, PP_{min}, and PP_{unit} depends on the architecture, and the values used in this paper were selected after experiments with a range of values. To decide whether proxying is appropriate, there has to be an extra check before each `read-request` is issued by a client x to a home node y:

if $[LB_{(x,y)} > 0]$ **and** $[(PP_{(x,y)} \times PP_{unit}) > (T_{curr} - LB_{(x,y)})]$
then send a `proxy-read-request`,
otherwise send a normal `read-request`.

3 Simulated Architecture and Experimental Design

The cc-NUMA design which is simulated for this work has already been described in [7], so this section concentrates on the changes required to support adaptive proxies and alternative strategies for caching proxied data. The caches are kept coherent using an invalidation-based, distributed directory protocol using singly-linked lists [9]. The benchmark applications are summarised in Table 1. GE implements a Gaussian Elimination algorithm [2]. CFD is a computational fluid dynamics application modelling laminar flow [8]. The remaining six applications were taken from Stanford's SPLASH-2 suite [10].

The adaptive proxies scheme adjusts the proxying period according to the level of congestion at individual home nodes. However it has the storage overheads of holding the $LB_{(x,y)}$, $PP_{(x,y)}$, PP_{unit}, PP_{max}, and PP_{min} values at each node. There are also the processing overheads of adjusting $PP_{(x,y)}$, and checking before issuing each remote read-request.

Implementing a separate proxy buffer would require a node controller which is capable of using a small area of the local memory for its own purposes (*e.g.* [4]), or which has some dedicated storage within the node controller (similar to [5]).

4 Experimental Results

This section presents the results obtained from execution-driven simulations of the adaptive proxy strategy using the three proxy data caching policies[1]. The results are summarised in Table 2, and are presented in terms of relative speedup, *i.e.* the ratio of the execution time for the fastest algorithm running on one processor to the execution time on $\mathcal{P}$ processors. For proxy caching in the SLC, the read-requests benefit from being spread around the system during the proxying period. However the scheme suffers from over-using proxies for the Ocean-Contig application (cache pollution and too large a value for PP_{unit}), and so has no

[1] A detailed analysis of the simulation results can be found in [6].

Table 1. Benchmark applications

application	problem size	application	problem size
Barnes	16K particles	GE	512×512 matrix
CFD	64×64 grid	Ocean-Contig	258×258 ocean
FFT	64K points	Ocean-Non-Contig	258×258 ocean
FMM	8K particles	Water-Nsq	512 molecules

overall balance point for the eight benchmark applications[2]. The GE application exhibits some bottleneck problems when $\mathcal{NPC}$=1&2, where proxy messages are sent to an already congested node, leading to a rise in overall queueing delay (although this is compensated for by the gains at other nodes).

The non-caching proxy policy results show that the proxying technique is still effective even when the opportunities for combining are restricted. The balance point at $\mathcal{NPC}$=1 occurs both because the chances of combining are greatest (since there is only one proxy node for a given data line), and because the Ocean-Contig application is able to benefit from the reduced cache pollution.

With a separate proxy buffer, there are three balance points, at $\mathcal{NPC}$=2,6,&7. The proxy buffer technique avoids the cache interference patterns seen with SLC caching for Barnes and Ocean-Non-Contig, while keeping most of the benefits of combining (unlike the non-caching approach). Ocean-Non-Contig in particular, which has poor data locality, benefits from the reduction in SLC cache pollution and the combining of proxy read requests. The results for Ocean-Contig highlight a subtle side-effect of using proxies. For values of $\mathcal{NPC} \geq 1$ the performance is determined by the effect of the use of proxies on the overall barrier delay. The changes in barrier delay result from redistributing messages to proxy nodes and the delays experienced by other messages queueing for service at proxy nodes.

Overall the adaptive proxy scheme gets the best performance with the separate proxy buffer, obtaining balance points at $\mathcal{NPC}$=2,6&7. A balance point is more desirable than a value of $\mathcal{NPC}$ which gives the best result for a specific application because we aim to get reasonable performance for a wide range of applications without the need to tune applications to suit the system. However, the no-proxy-caching strategy (when $\mathcal{NPC}$=1 to maximise combining) is a reasonable solution where it is not possible to have proxy buffers.

5 Conclusions and Further Work

This paper has proposed adaptive proxies to alleviate the performance problems arising from read accesses to widely-shared data. The simulation results show that adaptive proxies (with a separate proxy buffer or with no-cache-proxies)

[2] A balance point is where the partition into $\mathcal{NPC}$ proxy clusters results in improved performance for all eight benchmark applications.

Table 2. Benchmark relative speedups with a separate proxy buffer (64 processors)

application	relative speedup no proxies	proxy caching method	% change in execution time (+ is better, − is worse) for $\mathcal{NPC}$ = 1 to 8							
			1	2	3	4	5	6	7	8
Barnes	46.3	SLC	+0.1	+3.2	+0.4	+0.4	+0.4	+0.2	-0.1	+0.2
		none	+0.4	+3.7	0.0	0.0	+0.5	+0.3	+0.1	+0.2
		buffer	0.0	+3.3	+0.4	+0.2	+0.2	+0.2	+0.4	+0.4
CFD	28.3	SLC	+9.2	+13.1	+11.3	+11.6	+11.2	+10.4	+10.6	+12.1
		none	+12.9	+13.7	+13.6	+12.7	+12.9	+13.5	+12.7	+12.5
		buffer	+9.4	+9.4	+9.0	+12.5	+10.7	+10.8	+10.5	+12.7
FFT	47.3	SLC	+11.9	+11.6	+11.3	+11.4	+11.2	+11.5	+11.0	+11.0
		none	+11.7	+11.2	+11.3	+11.4	+11.3	+11.1	+11.2	+10.8
		buffer	+11.9	+11.9	+11.6	+11.8	+11.4	+11.4	+11.0	+10.8
FMM	52.4	SLC	+0.4	+0.4	+0.4	+0.4	+0.4	+0.5	+0.4	+0.4
		none	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4	+0.4	+0.4
		buffer	+0.4	+0.3	+0.4	+0.4	+0.5	+0.4	+0.4	+0.4
GE	21.6	SLC	+30.5	+30.7	+31.4	+31.2	+31.7	+31.6	+31.4	+31.6
		none	+30.3	+30.5	+31.4	+31.0	+31.3	+31.3	+31.0	+31.0
		buffer	+30.7	+30.9	+31.8	+31.3	+31.8	+31.8	+31.5	+31.7
Ocean-Contig	49.7	SLC	-1.3	-2.8	-6.1	-3.5	-1.4	-3.6	-0.4	-3.6
		none	+3.2	+0.5	-1.0	-2.3	0.0	-2.6	-0.1	-1.1
		buffer	-2.4	+1.5	-1.5	-6.8	-0.2	+1.9	+0.8	-0.7
Ocean-Non-Contig	48.2	SLC	+7.8	+7.6	-6.3	+2.0	+4.1	+6.6	-8.3	-1.5
		none	+0.5	-3.6	+4.4	-11.3	+3.7	+4.7	+7.4	+3.3
		buffer	+4.5	+6.5	+5.8	+2.3	-0.2	+3.0	+3.7	+6.8
Water-Nsq	55.3	SLC	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2
		none	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.1	+0.2
		buffer	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2	+0.2

give stable performance, allow the programmer to write portable applications which are less "architecture specific", and save on performance tuning because the widely-shared data access bottleneck is dealt with automatically. To evaluate the commercial viability of adaptive proxies it would be necessary to investigate the performance effects of commercial workloads. Further work is also needed to assess the impact of different network topologies and processor cluster nodes, and alternative implementations of the proxy buffer.

Acknowledgements

This work was funded by the U.K. Engineering and Physical Sciences Research Council through a Research Studentship. We would also like to thank Ashley Saulsbury and Andrew Bennett for their work on the ALITE simulator.

References

[1] Craig Anderson and Anna R. Karlin. Two adaptive hybrid cache coherency protocols. In *the 2nd HPCA, San Jose, California*, pages 303–313, February 1996.

[2] Satish Chandra et al. Where is time spent in message-passing and shared-memory programs? *6th ASPLOS, in SIGPLAN Notices*, 29(11):61–73, October 1994.

[3] Chris Holt et al. The effects of latency, occupancy and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[4] Jeffrey Kuskin. *The FLASH Multiprocessor: designing a flexible and scalable system*. PhD thesis, Computer Systems Laboratory, Stanford University, November 1997. Also available as technical report CSL-TR-97-744.

[5] Maged Michael and Ashwini Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *the 5th HPCA, Orlando*, pages 142–151, January 1999.

[6] Sarah A. M. Talbot. *Shared-Memory Multiprocessors with Stable Performance*. PhD thesis, Department of Computing, Imperial College, London, June 1999. Available on-line from `http://www.doc.ic.ac.uk/~samt/pub.html`.

[7] Sarah A. M. Talbot and Paul H. J. Kelly. Reactive proxies: a flexible protocol extension to reduce ccNUMA node controller contention. In *Euro-Par 98*, volume 1470 of *LNCS*, pages 1062–1075. Springer-Verlag, September 1998.

[8] B. A. Tanyi. *Iterative Solution of the Incompressible Navier-Stokes Equations on a Distributed Memory Parallel Computer*. PhD thesis, UMIST, 1993.

[9] Manu Thapar and Bruce Delagi. Stanford distributed-directory protocol. *IEEE Computer*, 23(6):78–80, June 1990.

[10] Steven Cameron Woo et al. The SPLASH-2 programs: characterization and methodological considerations. *22nd ISCA, in Computer Architecture News*, 23(2):24–36, 1995.

Topic 09
Distributed Systems and Algorithms

Ernst W. Mayr

Local Chair

This topic deals with new developments in distributed systems and algorithms. The wide acceptance of the Internet standards and technologies makes it hard to imagine a situation in which it would be easier to argue about their importance than it is today.

Areas of interest include, but are not limited to:

- mobile computing
- distributed algorithms in telecommunications
- fault tolerance of distributed systems
- resource sharing in distributed systems
- openness in distributed systems
- concurrency, performance and scalability in distributed systems
- transparency in distributed systems
- design and analysis of distributed algorithms
- real-time distributed algorithms and systems

Out of nineteen submissions to this track, seven papers were accepted and are presented in two sessions. The first session, containing three papers altogether, starts out with a presentation analyzing balancing networks with antitokens allowed. It is of interest which properties are preserved under this generalized definition. A necessary and sufficient condition for these properties is presented. The second paper considers the standard Internet routing strategy (each intermediate node knows the next edge of a shortest path to the target node) in a modified scenario: "Liars", nodes which give bad advice, are allowed. Three different models are examined in the context of various topologies, giving interesting results. The final paper in this session proposes a method which enables the systematic design of complete exchange algorithms for a wide range of topologies, including meshes and tori. In several cases the new algorithm outperforms previously known approaches for a significant range of system parameters.

The second session consists of three papers. The first concerns a special application of permutation routing in mesh topologies to automated guided vehicles. Under the constraint that no more than two vehicles (resp. packages) meet at any time an $O(n \log n)$ algorithm is obtained. The next paper, entitled "Self-Stabilizing Protocol for Shortest Path Tree for Multi-cast Routing in Mobile Networks", again, as the second paper in the first session, deals with the standard Internet routing strategy, although in a different setting and with a different goal: The nodes of the network can move and a parallel algorithm is sought which

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 573–574, 2000.
© Springer-Verlag Berlin Heidelberg 2000

updates the routing tables using only local information at the nodes. Such an algorithm is presented and analyzed.

Replication of data in asynchronous distributed systems can increase reliability and availability drastically, but also involves complicated coordination issues. The third paper addresses the replica management problem in which processes can crash and recover. A new solution to this problem is described. The last paper is about timestamping algorithms which are used to capture the causal ordering or the concurrency of events in distributed computations. This paper introduces a formal framework on timestamping algorithms, by characterizing some conditions which they have to satisfy in order to capture causality.

We would like to express our sincere appreciation to the other members of the program committee for their invaluable help in the entire selection process. We would also like to thank the numerous reviewers for their precious time and effort.

A Combinatorial Characterization of Properties Preserved by Antitokens

Costas Busch[1], Neophytos Demetriou[2], Maurice Herlihy[1], and Marios Mavronicolas[2]

[1] Department of Computer Science,
Brown University,
Providence, RI 02912
{cb,herlihy}@cs.brown.edu
[2] Department of Computer Science,
University of Cyprus,
Nicosia CY-1678, Cyprus
cs97dn1@ucy.ac.cy, mavronic@cs.ucy.ac.cy

Abstract. *Balancing networks* are highly distributed data structures used to solve multiprocessor synchronization problems. Typically, balancing networks are accessed by tokens, and the distribution of the tokens on the network's output specify the property of the network. However, tokens represent increment operations only, and tokens alone are not adequate for synchronization problems that require decrement operations. For such kinds of problems, *antitokens* have been used to represent decrement operations. It has been shown that several kinds of balancing networks which satisfy the *step property*, *smoothing property*, and the *threshold property* for tokens alone, preserve their properties even when antitokens are introduced. A fundamental question that was left open was to characterize *all* the properties of balancing networks which are preserved under the introduction of antitokens. In this work, we provide such a simple combinatorial characterization for all the properties which are preserved when antitokens are introduced.

1 Introduction

Balancing networks were devised by Aspnes *et al.* [4] as a novel class of distributed data structures that provide highly-concurrent, low-contention solutions to a variety of multiprocessor synchronization problems.

A balancing network is constructed from elementary switches with p *input wires* and q *output wires*, called (p,q)-*balancers*. A (p,q)-balancer accepts a stream of tokens on its p input wires. The i-th token to enter the balancer leaves on output wire $i \bmod q$, where $i = 0, 1, \ldots$. One can think of a balancer as having a "toggle" state variable tracking which output wire the next token should exit from. A token traversal amounts to a *Fetch&Increment* operation to the toggle variable. This operation includes reading the current state of the toggle, which is the wire the token will exit from, and then setting the toggle

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 575–582, 2000.
© Springer-Verlag Berlin Heidelberg 2000

to point to the next output wire. The distribution of the tokens on the output wires of the balancer satisfies the *step property* (explained below).

A balancing network is an acyclic network of balancers where output wires of some balancers are linked to input wires of other balancers (balancing networks look like sorting networks [15]). The network's *input wires* are those input wires of balancers not linked to any other balancer, and similarly for the network's *output wires*. Tokens enter the network on the input wires, typically several per wire, propagate asynchronously through the balancers and leave from the output wires, typically several per wire.

Balancing networks are classified according to the distribution of the exiting tokens on the output wires. In particular, *Counting networks* [4] are those balancing networks on which the exiting tokens satisfy the *step property*: the exiting tokens are distributed uniformly among the output wires and any excess tokens appear on the upper wires. On *smoothing networks* [1, 4] the output tokens satisfy the K*-smoothing property*: the sum of tokens on any two output wires differ by at most K. On *threshold networks* [4, 7] the output sequence satisfies the *threshold property*: the number of tokens on the bottom wire is increased by one for every bunch of w tokens, where w is the number of output wires of the network.

Based on balancing networks, simple and elegant algorithms have been developed to solve a variety of synchronization problems that appear in distributed computing systems. For example, counting networks are used to implement efficient distributed *Fetch&Increment* counters as well as *linearizable* counters [11]. Furthermore, smoothing networks solve *load sharing* problems [17], and threshold networks provide solutions to *barrier synchronization* problems [9]. For applications of balancing networks see [4, 11, 12, 14, 16].

A limitation of balancing networks is that they are accessed by tokens only. A token can be thought of as an "increment" operation issued by the process which inserts the token in the network. Using tokens only, the capabilities of balancing networks are limited to the use of only increment operations. However, many distributed algorithms require the ability to "decrement" shared objects as well. For example, the classical synchronization constructs of *semaphores* [8], *critical regions* [13], and *monitors* [10] all rely on applying both increment and decrement operations on shared counters.

In order to solve such kinds of problems Shavit and Touitou [16] invented the *antitoken*, an entity that a processor shepherds through the network in order to perform a decrement operation. Unlike a token, which traverses a balancer by fetching the toggle value and then advancing it, an antitoken sets the toggle back and then fetches it. Informally, an antitoken "cancels" the effect of the most recent token on the balancer's toggle state, and vice versa. Furthermore, when an antitoken and a token meet while they traverse a network they can "eliminate" each other without needing to traverse the rest of the network.

In the same paper, Shavit and Touitou provide an operational proof that a specific kind of counting networks which have the form of binary trees count correctly even when they are traversed by both tokens and antitokens. Namely, they

show that in these networks the step property is preserved by the introduction of antitokens.

Subsequently, Aiello *et al.* [2] generalized the results of Shavit and Touitou [16] to far more general classes of balancing networks and properties of balancing networks. More specifically, Aiello *et al.* considered *boundedness properties,* a generalization of the step and K-smoothing properties. They showed that boundedness properties are preserved by the introduction of antitokens. Busch *et al.* [5] considered the *threshold property* and they showed that this property is also preserved by the introduction of antitokens.

A fundamental question that was left open by the results in [2, 5, 16], is to formally characterize *all* properties of balancing networks that are preserved under the introduction of antitokens. In this work, we provide the first answer to this fundamental question.

We provide a simple, combinatorial characterization for all properties of balancing networks which are preserved when antitokens are introduced. In particular, for any arbitrary balancing network, we define a new, natural class of properties, that we call *closed under the nullity of the balancing network*, which precisely characterizes all the properties preserved by antitokens. This characterization provides necessary and sufficient conditions for all the properties that are preserved. For any property that is satisfied by a balancing network for tokens only, then our characterization implies that this property is preserved when antitokens are introduced if and only if the property is closed under the nullity of the network.

The combinatorial characterization provides a theoretical tool for identifying which properties are preserved by the introduction of antitokens. For example, consider some property of a balancing network for which we know the network satisfies for tokens. In order to prove that this property will be preserved by the antitokens we only need to show that the property is closed under the nullity of the network. Having this theoretical tool, the practitioner can identify if a specific property of balancing can be used to implement algorithms that require decrements.

Moreover, the necessary condition of the characterization enables us to classify all the properties for which we already know are preserved with antitokens. This necessary condition simply says that all these properties must satisfy the characterization and therefore are closed under the nullity of a balancing network. Consequently, from the results of [2], we can infer that the the step property, the K-smoothing property, and in general the boundedness property are all closed under the nullity of a balancing network. Furthermore, from the results of [5], we can infer that the threshold property is closed under the nullity of a balancing network.

The rest of this paper is organized as follows. Section 2 provides some necessary background. In section 3 we describe properties of balancing networks. We present our main combinatorial characterization result in Section 4. We give our conclusions in Section 5.

2 Framework

Throughout our discussion we consider integer vectors. For any integer $g \geq 2$, $\mathbf{x}^{(g)}$ denotes the vector $\langle x_0, x_1, \ldots, x_{g-1} \rangle^{\mathrm{T}}$. For any vector $\mathbf{x}^{(g)}$, denote $\|\mathbf{x}^{(g)}\|_1 = \sum_{i=0}^{g-1} x_i$. We use $\mathbf{0}^{(g)}$ to denote $\langle 0, 0, \ldots, 0 \rangle^{\mathrm{T}}$, a vector with g zero entries. In a *constant vector* all entries are equal to some constant c. We say that a vector is *non-negative*, if all of its entries are non-negative integers. We say that an integer d *divides* a vector $\mathbf{x}^{(g)}$ if each entry of $\mathbf{x}^{(g)}$ is some integer multiple of d.

For the rest of our discussion, we consider balancers and balancing networks in *quiescent configurations* in which no tokens and antitokens are traversing the network, namely, all the tokens and antitokens that have ever entered the balancer have left it.

We think of a token as a positive unit +1, and the antitoken as a negative unit -1. Consider an $(f_{\mathrm{in}}, f_{\mathrm{out}})$-balancer. For each input index i, $0 \leq i < f_{\mathrm{in}}$, we denote by x_i the algebraic sum of tokens and antitokens that have entered on input wire i; that is, x_i is the number of tokens *minus* the number of antitokens that have entered on input wire i. We say that the vector $\mathbf{x}^{(f_{\mathrm{in}})} = \langle x_0, x_1, \ldots, x_{f_{\mathrm{in}}-1} \rangle^{\mathrm{T}}$ is an *input vector* of the balancer. Similarly, we define the *output vector* of the balancer. In the same way, we define input and output vectors for balancing networks. Note that when we are considering tokens only the input and output vectors are non-negative. Vectors can take negative values only when we consider antitokens too.

Let $\mathcal{B}$ be a balancing network with w_{in} input wires and w_{out} output wires. We call w_{in} the fan-in, and w_{out} the fan-out of the network. Take any input vector $\mathbf{x}^{(w_{\mathrm{in}})}$ to $\mathcal{B}$ and let $\mathbf{y}^{(w_{\mathrm{out}})}$ be the corresponding output vector. For each input vector $\mathbf{x}^{(w_{\mathrm{in}})}$, there is a unique output vector $\mathbf{y}^{(w_{\mathrm{out}})}$, and this allows us to treat the network $\mathcal{B}$ as a function on vectors, and we write $\mathcal{B}(\mathbf{x}^{(w_{\mathrm{in}})}) = \mathbf{y}^{(w_{\mathrm{out}})}$. We write also $\mathcal{B} : \mathbf{x}^{(w_{\mathrm{in}})} \rightarrow \mathbf{y}^{(w_{\mathrm{out}})}$ to denote the network $\mathcal{B}$.

Clearly, $\mathcal{B}(\mathbf{0}^{(w_{\mathrm{in}})}) = \mathbf{0}^{(w_{\mathrm{out}})}$. In any quiescent configuration it holds that $\|\mathcal{B}(\mathbf{x}^{(w_{\mathrm{in}})})\|_1 = \|\mathbf{x}^{(w_{\mathrm{in}})}\|_1$, which means that the algebraic sum of tokens and antitokens that have entered the network is the same with the algebraic sum of tokens and antitokens that have left the network. This also includes the tokens and antitokens that have been "eliminated" in the network, since their algebraic sum is zero.

Consider now an $(f_{\mathrm{in}}, f_{\mathrm{out}})-$balancer b. The *state* of balancer b, on input sequence $\mathbf{x}^{(f_{\mathrm{in}})}$, is defined to be $\mathrm{state}_b(\mathbf{x}^{(f_{\mathrm{in}})}) = \|\mathbf{x}^{(f_{\mathrm{in}})}\|_1 \bmod f_{\mathrm{out}}$. We remark that the state of balancer b is some integer in the set $\{0, 1, \ldots, f_{\mathrm{out}} - 1\}$, which captures the "position" to which the balancer is set as a toggle mechanism. Consider now a balancing network $\mathcal{B} : \mathbf{x}^{(w_{\mathrm{in}})} \rightarrow \mathbf{y}^{(w_{\mathrm{out}})}$. The *state* of $\mathcal{B}$, denoted $\mathrm{state}_{\mathcal{B}}(\mathbf{x}^{(w_{\mathrm{in}})})$, is defined to be the collection of the states of its individual balancers. The *initial state* of network $\mathcal{B}$ is the state $\mathrm{state}_{\mathcal{B}}(\mathbf{0}^{(w_{\mathrm{in}})})$.

In respect to the state of a balancing network, Aiello *et al.* [2] have defined *fooling pairs* and *null vectors* as follows. Say that two input vectors $\mathbf{x}_1^{(w_{\mathrm{in}})}$ and $\mathbf{x}_2^{(w_{\mathrm{in}})}$ are *a fooling pair to network* $\mathcal{B} : \mathbf{x}^{(w_{\mathrm{in}})} \rightarrow \mathbf{y}^{(w_{\mathrm{out}})}$, if $\mathrm{state}_{\mathcal{B}}(\mathbf{x}_1^{(w_{\mathrm{in}})}) = \mathrm{state}_{\mathcal{B}}(\mathbf{x}_2^{(w_{\mathrm{in}})})$. Roughly speaking, a fooling pair "drives" all balancers of the

network to identical states. Say that $\mathbf{x}^{(w_{\text{in}})}$ is a *null vector to network* $\mathcal{B}$ if the vectors $\mathbf{x}^{(w_{\text{in}})}$ and $\mathbf{0}^{(w_{\text{in}})}$ are a fooling pair to $\mathcal{B}$. Intuitively, a null vector "drives" the network back to its initial state. Using results on properties of fooling pairs and null vectors from Aiello *et al.* [2], it is straightforward to obtain the following "linearity" lemma for null vectors.

Lemma 1. *Consider a balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \to \mathbf{y}^{(w_{\text{out}})}$. *Take any input vector* $\mathbf{x}^{(w_{\text{in}})}$ *and any null vector* $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ *to* $\mathcal{B}$. *Then,*

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} \pm \tilde{\mathbf{x}}^{(w_{\text{in}})}) = \mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \pm \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}).$$

For any balancing network $\mathcal{B}$, denote by $W_{\text{out}}(\mathcal{B})$ the product of the fan-outs of balancers of $\mathcal{B}$. Aiello *et al.* [2] show the following.

Lemma 2 (Aiello et al. [2]). *Consider a balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \to \mathbf{y}^{(w_{\text{out}})}$. *Assume that* $W_{\text{out}}(\mathcal{B})$ *divides* $\mathbf{x}^{(w_{\text{in}})}$. *Then,* $\mathbf{x}^{(w_{\text{in}})}$ *is a null vector to* $\mathcal{B}$.

3 Properties

A property $\mathbf{\Pi}$ is a (computable) predicate on integer vectors. We identify $\mathbf{\Pi}$ with the set of (integer) vectors satisfying it. Say that *a vector* $\mathbf{y}^{(w_{\text{out}})}$ *has the property* $\mathbf{\Pi}$ if $\mathbf{y}^{(w_{\text{out}})}$ satisfies $\mathbf{\Pi}$. Say that *a balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \to \mathbf{y}^{(w_{\text{out}})}$ *has a property* $\mathbf{\Pi}$, if all output vectors $\mathbf{y}^{(w_{\text{out}})}$ have the property $\mathbf{\Pi}$, for any input vectors (not only non-negative input vectors). Below we describe in details several interesting properties.

Boundedness properties were introduced by Aiello *et al.* [2]. Fix any integer $g \geq 2$. For any integer $K \geq 1$, the K*-smoothing property* [1] is defined to be the set of all vectors $\mathbf{y}^{(g)}$ such that for any entries y_j and y_k of $\mathbf{y}^{(g)}$, where $0 \leq j, k < g$, it holds $|y_j - y_k| \leq K$. A *boundedness property* is any subset of some K-smoothing property, and this subset is closed under addition with a constant vector, for some integer $K \geq 1$. Thus, a boundedness property is a strict generalization of the smoothing property. Clearly, there are infinitely many boundedness properties. The *step property* [4] is defined to be the set of all vectors $\mathbf{y}^{(g)}$ such that for any entries y_j and y_k of $\mathbf{y}^{(g)}$, where $0 \leq j < k < g$, it holds $0 \leq y_j - y_k \leq 1$. Clearly, the step property is a boundedness property, since any vector that has the step property, has also the 1-smoothing property (but not vice versa). The main result of Aiello *et al.* [2] establishes that allowing negative inputs (antitokens) does not spoil the boundedness property of a balancing network.

Theorem 1 (Aiello et al. [2]). *Fix any boundedness property* $\mathbf{\Pi}$. *Consider any balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \to \mathbf{y}^{(w_{\text{out}})}$ *such that* $\mathbf{y}^{(w_{\text{out}})}$ *has the boundedness property* $\mathbf{\Pi}$ *whenever* $\mathbf{x}^{(w_{\text{in}})}$ *is a non-negative input vector. Then,* $\mathcal{B}$ *has the boundedness property* $\mathbf{\Pi}$.

The *threshold property* [4, 7] is the set of all vectors $\mathbf{y}^{(g)}$, such that for the entry y_{g-1} of $\mathbf{y}^{(g)}$, it holds $y_{g-1} = \lfloor \|\mathbf{y}^{(g)}\|_1 / g \rfloor$. It has been observed in [5] that

the threshold property is not a boundedness property in all non-trivial cases (where $g > 2$). Thus, Theorem 1 does not apply a fortiori to this property. The main result of Busch *et al.* [5] establishes that allowing negative inputs (antitokens) does not spoil the threshold property of a balancing network.

Theorem 2 (Busch et al. [5]). *Consider any balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \rightarrow \mathbf{y}^{(w_{\text{out}})}$ *such that* $\mathbf{y}^{(w_{\text{out}})}$ *has the threshold property whenever* $\mathbf{x}^{(w_{\text{in}})}$ *is a non-negative vector. Then,* $\mathcal{B}$ *has the threshold property.*

4 Combinatorial Characterization

A fundamental question that was left open by the results in Theorems 1 and 2, is to formally characterize all the properties of balancing networks that are preserved under the introduction of antitokens. In this section, we give such a combinatorial characterization as follows.

Definition 1. *Consider any balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \rightarrow \mathbf{y}^{(w_{\text{out}})}$*. A property* $\mathbf{\Pi}$ *is* closed under the nullity of $\mathcal{B}$ *if for all non-negative input vectors* $\mathbf{x}^{(w_{\text{in}})}$ *and for all non-negative null vectors* $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ *to* $\mathcal{B}$*, it holds that* $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \in \mathbf{\Pi}$ *implies* $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \pm B(\tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}$.

The use of non-negative vectors in the above definition allows us to determine whether any given property of a balancing network is closed under the nullity of the network by examining how the network behaves for tokens only.

In the next claim we establish our main result. We show that being closed under the nullity of a balancing network is a necessary and sufficient condition for the property to be preserved under the introduction of antitokens.

Theorem 3. *Fix a property* $\mathbf{\Pi}$*. Consider any balancing network* $\mathcal{B} : \mathbf{x}^{(w_{\text{in}})} \rightarrow \mathbf{y}^{(w_{\text{out}})}$ *such that* $\mathbf{y}^{(w_{\text{out}})} \in \mathbf{\Pi}$ *whenever the input vector* $\mathbf{x}^{(w_{\text{in}})}$ *is non-negative. Then,* $\mathcal{B}$ *has the property* $\mathbf{\Pi}$ *if and only if* $\mathbf{\Pi}$ *is closed under the nullity of* $\mathcal{B}$*.*

Proof. First, we prove the "if" direction of the claim. Consider any arbitrary input vector $\mathbf{x}^{(w_{\text{in}})}$. We will show that $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \in \mathbf{\Pi}$.

Construct from $\mathbf{x}^{(w_{\text{in}})}$ an non-negative input vector $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ such that for each index i, $\tilde{x_i}$ is the least positive multiple of $W_{\text{out}}(\mathcal{B})$ so that $0 \leq x_i + \tilde{x_i}$. Clearly, the vector $\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}$ is non-negative. Furthermore, $W_{\text{out}}(\mathcal{B})$ divides $\tilde{\mathbf{x}}^{(w_{\text{in}})}$, and from Lemma 2 it follows that $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ is a null vector.

By applying Lemma 1 with vectors $\mathbf{x}^{(w_{\text{in}})}$ and $\tilde{\mathbf{x}}^{(w_{\text{in}})}$, we obtain

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}) = \mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) + \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}),$$

so that

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) = \mathcal{B}(\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}) - \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}).$$

Since the vector $\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}$ is non-negative, we have by assumption that

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}.$$

Furthermore, since property $\mathbf{\Pi}$ is closed under the nullity of $\mathcal{B}$, and since $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ is a non-negative null vector, we have by Definition 1 that

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} + \tilde{\mathbf{x}}^{(w_{\text{in}})}) - \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}.$$

Subsequently, $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \in \mathbf{\Pi}$, as needed.

We continue to show the "only if" part of the claim. Take any non-negative input vector $\mathbf{x}^{(w_{\text{in}})}$ to $\mathcal{B}$ and any non-negative null vector $\tilde{\mathbf{x}}^{(w_{\text{in}})}$ to $\mathcal{B}$. Trivially, $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \in \mathbf{\Pi}$, and thus, by Definition 1, we only need to show that $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \pm \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}$. By Lemma 1,

$$\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} \pm \tilde{\mathbf{x}}^{(w_{\text{in}})}) = \mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \pm \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}).$$

Obviously, $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})} \pm \tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}$. Subsequently, $\mathcal{B}(\mathbf{x}^{(w_{\text{in}})}) \pm \mathcal{B}(\tilde{\mathbf{x}}^{(w_{\text{in}})}) \in \mathbf{\Pi}$, as needed.

Since the boundedness and the threshold properties were shown in [2, 5] to be preserved under the introduction of antitokens (see also Theorems 1 and 2), the necessary condition of Theorem 3 implies that these properties are closed under the nullity of any balancing network.

The sufficient condition of Theorem 3 can be used to determine if any given property is preserved with antitokens. In general, we are given a property $\mathbf{\Pi}$ which we know it is satisfied by a balancing network when the network is accessed by tokens only. We want to find out if this property will still be preserved even when the network is accessed by antitokens too. In order to show this, the sufficient condition of Theorem 3 implies that we only need to prove that the property is closed under the nullity of the network.

We can strengthen Definition 1 and Theorem 3 so that in their statements, the non-negative input vectors and null vectors, are restricted to vectors with entries in the range $[0, W_{\text{out}}(\mathcal{B})]$. This way, we obtain a new verification procedure for identifying whether a particular network $\mathcal{B}$ satisfies a property closed under the nullity of a network. In particular, if property $\mathbf{\Pi}$ is closed under the nullity of a network $\mathcal{B}$ (for input vectors and null vectors with entries in the range $[0, W_{\text{out}}\mathcal{B}]$), Theorem 3 implies that in order to verify that $\mathcal{B}$ satisfies the property $\mathbf{\Pi}$, it suffices to verify that all vectors with entries in the interval $[0, W_{\text{out}}(\mathcal{B})]$ satisfy $\mathbf{\Pi}$. We can simply feed all these vectors to the network and examine if each respective output vector satisfies $\mathbf{\Pi}$. This is the *first* verification procedure established for properties satisfied by balancing networks that are traversed by both tokens and antitokens. (For more about verification algorithms see [4, 6].)

5 Conclusion

We have provided a combinatorial characterization of the properties satisfied by balancing networks traversed by tokens alone that are preserved when antitokens are introduced. Our results close the main problem left open by the results in [2, 5]. An interesting question still left open by our work is to provide a corresponding characterization for *randomized* balancing networks [3], where the balancers distribute the tokens on their output wires following some random permutation.

References

[1] E. Aharonson and H. Attiya, "Counting Networks with Arbitrary Fan-Out," *Distributed Computing,* Vol. 8, pp. 163–169, 1995.

[2] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit, and D. Touitou, "Supporting Increment and Decrement Operations in Balancing Networks," *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science,* G. Meinel and S. Tison eds., pp. 377–386, Vol. 1563, Lecture Notes in Computer Science, Springer-Verlag, Trier, Germany, March 1999. Also, to appear in the *Chicago Journal of Theoretical Computer Science.*

[3] W. Aiello, R. Venkatesan and M. Yung, "Coins, Weights and Contention in Balancing Networks," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 193–205, Los Angeles, California, August 1994.

[4] J. Aspnes, M. Herlihy and N. Shavit, "Counting Networks," *Journal of the ACM,* Vol. 41, No. 5, pp. 1020–1048, September 1994.

[5] C. Busch, N. Demetriou, M. Herlihy and M. Mavronicolas, "Threshold Counters with Increments and Decrements," *Proceedings of the 6th International Colloquium on Structural Information and Communication Complexity,* pp. 47–61, Lacanau, France, July 1999.

[6] C. Busch and M. Mavronicolas, "A Combinatorial Treatment of Balancing Networks," *Journal of the ACM,* Vol. 43, No. 5, pp. 794–839, September 1996.

[7] C. Busch and M. Mavronicolas, "Impossibility Results for Weak Threshold Networks," *Information Processing letters,* Vol. 63, No. 2, pp. 85–90, July 1997.

[8] E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages,* pp. 43–112, Academic Press, 1968.

[9] D. Grunwald and S. Vajracharya, "Efficient Barriers for Distributed Shared Memory Computers," *Proceedings of the 8th International Parallel Processing Symposium,* IEEE Computer Society Press, April 1994.

[10] P. B. Hansen, *Operating System Principles,* Prentice Hall, Englewood Cliffs, NJ, 1973.

[11] M. Herlihy, B.-C. Lim and N. Shavit, "Concurrent Counting," *ACM Transactions on Computer Systems,* Vol. 13, No. 4, pp. 343–364, 1995.

[12] M. Herlihy, N. Shavit and O. Waarts, "Linearizable Counting Networks," *Distributed Computing,* Vol. 9, pp. 193–203, 1996.

[13] C. A. R. Hoare and R. N. Periott, *Operating Systems Techniques,* Academic Press, London, 1972.

[14] S. Kapidakis and M. Mavronicolas, "Distributed, Low Contention Task Allocation," *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing,* pp. 358–365, New Orleans, Louisiana, October 1996.

[15] D. E. Knuth, "The Art of Computer Programming III: Sorting and Searching," Vol. 3, Addison-Wesley, 1973.

[16] N. Shavit and D. Touitou, "Elimination Trees and the Construction of Pools and Stacks," *Theory of Computing Systems,* Vol. 30, No. 6, pp. 545–570, November/December 1997.

[17] S. Zhou, X. Zheng, J. Wang and P. Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *Software–Practice and Experience,* Vol. 23, No. 12, pp. 1305–1336, December 1993.

Searching with Mobile Agents in Networks with Liars

Nicolas Hanusse[1], Evangelos Kranakis[2], and Danny Krizanc[3]

[1] INRIA Rocquencourt, Carleton University, School of Computer Science, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, Canada,
hanusse@scs.carleton.ca
[2] Carleton University,
kranakis@scs.carleton.ca
[3] Weysleyan University, Middletown, Connecticut 06459, US,
dkrizanc@caucus.cs.wesleyan.edu

Abstract. In this paper, we present algorithms to search for an item s contained in a node of a network, without prior knowledge of its exact location. Each node of the network has a database that will answer queries of the form "how do I get to s?" by responding with the first edge on a shortest path to s. It may happen that some nodes , called *liars*, give bad advice. If the number of liars k is bounded, we show different strategies to find the item depending on the topology of the network. In particular we consider the complete graph, ring, torus, hypercube and bounded degree trees.

1 Introduction

Mobile agents can perform very complex information gathering, like assembling and digesting "related" topics of interest. Depending on their "behavior" mobile agents can be classified as reactive (responding to changes in their environment) or pro-active (seeking to fulfill certain goals). Moreover agents may choose to remain stationary (filtering incoming information) or become mobile (searching for specific information across the Internet and retrieving it) [16]. There are numerous examples of such agents in use today, including the Internet search engines, like Yahoo, Lycos, etc.

In the present paper we consider the problem of searching for an item in a distributed network in the presence of "liars." The objective is to design a mobile agent that travels along the network links in order to locate the item. Although the location of the item in the network is initially unknown, information about its whereabouts can be obtained by querying the nodes of the network. The nodes have databases providing the first edge on a shortest path to the item sought. The agent queries the nodes; the queried nodes respond either by providing a link adjacent to them that is on a shortest path to the node that holds the item or if the desired item is at the node itself then the node answers by providing

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 583–590, 2000.
© Springer-Verlag Berlin Heidelberg 2000

it to the agent. However certain nodes in the network may be liars, e.g., due to out-of-date network information in their databases. The liars are unknown to the mobile agent that must still find the item despite the fact that queries to responses may be wrong. In this paper we give deterministic algorithms for searching in a distributed network with a bounded number of liars that has the topology of a complete network, ring, torus, hypercube, or trees under three models of liars.

A variant of the above *searching* model, was introduced in [9], where the network topologies considered were the *ring* and the *torus* and the nodes respond to queries with a bounded probability of being incorrect. Additional investigations under the same model of "searching with uncertainty" were carried out for fully interconnected networks in [10]. Models with faulty information in the nodes have been considered before for the problem of routing (see [1, 3, 7, 8, 14]). However, in this problem it is assumed that the identity of the node that contains the information is known, and what is required is to reach this node following the best possible route. Search problems in graphs, where the identity of the node that contains the information sought is not known, have been considered before. These include *deterministic search games*, where a fugitive that possesses some properties hides in the nodes or edges of a graph [5, 12, 13]), and the problem of exploring an *unknown* graph [2, 11, 15]. Our model is similar in spirit to the model in [4] where the authors propose algorithms to search for a point on a line or on a lattice drawn on the plane. While in the models they consider there is limited if any knowledge of the location of the point, the nodes along the way do not provide new location information at each step as in our case.

1.1 Preliminaries and Definitions

In order to present the problem more precisely, we must define the search model in a given network. Since a mobile agent does not know if a network has liars, we suppose it assumes the number of liars is bounded by k. This assumption may affect the moves of the mobile agent. Thus, the complexity of our results will depend on the actual distance d of the mobile agent to the destination as well as the number of liars assumed.

The mobile agent is basically a software program running an algorithm that requires a certain amount of memory, storing relevant information about its current position in the network, e.g. in a binary tree the distance to the root, in a ring the distance from the starting node, etc. We will see later that the algorithm depends on the topology of the network and we will consider different trade-offs between the amount of memory required by the mobile agent and the number of steps, i.e the number of moves of the mobile agent.

A network of n nodes is represented as a connected undirected graph $G = (V, E)$ where V is the set of vertices or nodes and E the set of edges or links. Let s denote the item the mobile agent is searching for and assume there is a unique node in G containing s. A *query* $Q_u(s)$ returns either s if the node u contains

s or a subset of edges, incident to u, belonging to a shortest path leading to the item s. If $Q_u(s)$ returns an edge that does not belong to a shortest path to s, the node u is called a *liar*, otherwise a *truthteller*. The path $p = u_0 u_1 \cdots u_\alpha$ is a sequence of nodes followed by the mobile agent until item s is found. The number of edges followed by the path p is called the *number of steps* of the mobile agent, which is denoted by α. If there are no liars we expect that the mobile agent will follow an optimal path, i.e. if $k = 0$, it is obvious that $\alpha = d$ where d is the distance between the starting node of the mobile agent and the node containing the item. δ (resp. Δ) is the minimal (resp. maximal) degree of a given network. By convention, we consider that the nodes can be labelled by the set $\{1, 2, \ldots, n\}$.

1.2 Models

We consider three models of responses to queries:

One advice per node with co-ordination (CO) Model: In this case, a query returns a unique edge. We assume some preprocessing was done when building the databases stored in each node u of V. Let v be the node containing s and choose a fixed shortest path tree with destination v. For a given node u, $Q_u(s) = e$ where e is the (unique) outgoing edge incident to u chosen in this shortest path tree. If a node indicates an edge on another shortest path this node is considered to be a liar. The mobile agent is assumed to have knowledge as to how the shortest path trees were originally constructed. For example, we assume they always report first a row and then a column in the case of the torus. The truthtellers are *co-ordinated* in that the set of edges they report leads to the construction of a particular shortest path spanning tree. An adversary may decide which nodes are liars but has no influence over which edges are to be reported by the truthtellers.

One advice per node without co-ordination (NCO) Model: In this model an adversary does not only decide which nodes are liars but also which correct edge the truthtellers will report whenever there is a choice of shortest path edges.

One advice per edge (ECO) Model: In this model a truthteller returns $Q_u(s)$ equal to the set of all incident edges to u belonging to a shortest path tree. A liar may return any (presumably non-empty) subset of the edges incident to u. Again the adversary has no input as to what is returned by a truthteller.

1.3 Results

In this paper, we consider searching for an item under the above models and for different topologies: complete graph, ring, torus, hypercube, trees. In each case, we assume that the mobile agent knows the topology of the network and suspects a bounded number k of liars. We assume that the responses of the nodes are set before the start of the algorithm according to the model considered and that

they do not change throughout the running of the algorithm. The cost measures we consider for a given algorithm is the number of steps (i.e, edges traversed) and the amount of memory required by the mobile agent. Proofs are left out and only sketches of algorithms are presented due to space limitations.

2 Complete Graphs

In this section, we present two algorithms. The first one prioritizes the number of steps and the second one the amount of memory. We also establish two lower bounds on the number of steps for the complete graph and for any graph. Algorithm SearchComplete(s) works as follow: starting from a node u , we follow its advice to node u' unless we have already visited u' in which case we select any node not previously visited and go there.

Theorem 1. *In any complete graph of n vertices with k liars, a mobile agent can find an item in at most $k+1$ steps with $k \log n$ bits of memory .*

Theorem 2. *Let $D(u,p)$ be the set of nodes at a distance p from u and B_p be the set of nodes at a distance at most p. For any graph so that $|D(u,p)| > 1, |B_p| \leqslant k$ and $|B_{p+1}| > k$ then a mobile agent starting from u may require at least $d+k$ steps to find an item, where d is the distance between the starting node and s.*

We may be interested in a trade-off between the memory and the number of steps required by a mobile agent to find an item. Algorithm SearchComplete2 illustrates this idea: follow advice of nodes labeled $1, 2, \ldots, k+1$ until you find the item, i.e. if node labeled i gives a bad advice and sends you to a node u then go to node labeled $i+1$.

Theorem 3. *In any complete graph of n vertices and k liars, a mobile agent can find an item in at most $2k+3$ steps with $\log k$ bits of memory.*

3 Ring and Torus

For the ring, each vertex is of degree two and we may consider a global orientation known by each processor. Each node has a left and a right edge labelled respectively Left and Right , i.e. the query $Q_u(s)$ returns Left or Right. Algorithm SearchRing(s,k) works as follow: (1) choose a direction to follow, (2) move in this direction until either s is found or $k+1$ query responses in the opposite direction are given and then move in the opposite direction.

Theorem 4. *In a ring of n vertices with k liars, a mobile agent can find an item in at most $d+4k+2$ steps with $O(\log k)$ bits of memory.*

Theorem 5. *There exists a distribution of k liars in the ring of n vertices for which the number of steps is at least $d + 2k$.*

We present three algorithms to find the item in a torus of n vertices. As for the ring, we suppose there exists a global orientation of the edges known by each node and its four incident edges are labelled L, R, U, D for the left, right, up and down direction. We also use the notation $\leftarrow, \rightarrow, \uparrow, \downarrow$ for the edges. u represents the current location.If dir is a direction, $\bar{dir}$ indicates the opposite direction: $\leftarrow = \bar{\rightarrow}$ and $\uparrow = \bar{\downarrow}$. The *advice of a block* or of a rectangle consists in the set of directions $\{a_1, ..., a_t\}$ so that each a_i has been given at least $k + 1$ times.

CO Model: For this model, we assume truthtellers always report first a row and then a column. The algorithm SEARCHRINGII(s, m, l) travels in a set, called *block*, of l consecutive nodes along the direction m and returns the number of query responses for each direction $\leftarrow, \rightarrow, \uparrow, \downarrow$. The *advice of a block B* corresponds to the direction indicated by the majority of B.

The sketch of the algorithm SEARCHTORUS(S) is the following: (1) follow the advice of a block of size $2k + 1$ until two blocks B, B' are found with the opposite advice, (2) locate the column of s by a walk [1] in a square containing B, B',(3) find s in the column c using a search algorithm in a ring.

Theorem 6. *In any torus of n vertices and k liars, a mobile agent of $O(k \log k)$ bits of memory can find an item in at most $d + O(k)$ steps.*

NCO and ECO Models: In the NCO Model, the walk of SEARCHTORUS does not work. Indeed, a row of truthtellers may indicate different columns for the item. We propose a new strategy to choose a starting direction in SEARCHTORUSII, we make a search within a square of area $O(k)$ instead of a segment of $O(k)$ nodes along a given direction (in SEARCHTORUSIII, we will use the previous method). We propose a variant of an algorithm which can be found in [9] to choose a starting direction in a square:

SEARCHSQUARE(s, u, l): (a) For each direction dir, $a_{dir} = 0$, let $m = \{\}$; (b) mobile agent searches for the desired item s by testing all nodes in a square B of area l centered at node u; for each node of advice dir, $a_{dir} = a_{dir} + 1$; (c) return $\{a_{dir}\}$;

The idea of SEARCHTORUSII is the following: (1) we first locate s in a band of columns (or rows) $c_1, \ldots, c_w$ of width $w = O(\sqrt{k})$ finding two adjacent squares S, S' of area $4k + 1$ with different horizontal or vertical advice, (2) we find the vertical (horizontal) direction to follow by a walk in a rectangle R of size $O(\sqrt{k}) * (2k + 1)$ containing S, S' (3) we search for s in in the direction given by R in consecutive rectangles of size $O(\sqrt{k})$ in the direction given by R.

[1] this walk not described here finds c in $O(k)$ steps

Roughly speaking, since each iteration of Step 3[2] takes $O(k)$ steps and moves the mobile agent $\Omega(\sqrt{k})$ closer to the item, we have the following result:

Theorem 7. *In any torus of n vertices and k liars, a mobile agent can find an item in at most $O(d\sqrt{k})$ steps with $O(\log k)$ bits of memory.*

If $d = \Omega(\sqrt{k}\log k)$, another strategy illustrated by SearchTorusIII may be interesting: (1) we first locate s in a band of columns (or rows) $c_1, \ldots, c_w$ finding two consecutive blocks B, B' of size $4k+1$ with different advice[3], (2) The next steps consist of applying a variant of the dichotomy principle in rectangles of size $O(k) \times O(k)$ to find the column of s.

Theorem 8. *In any torus of n vertices and k liars, a mobile agent of $O(\log k)$ bits of memory can find an item in at most $O(d + k\log k)$ steps.*

In the ECO Model, the mobile agent may use the same algorithms as for CO Models. Indeed, the mobile agent can do the co-ordination itself choosing one edge per node. Since we have a lower bound of $d + \Omega(k)$ steps for any model, the upper bounds in ECO Model does not change a lot if we do not pay particular attention to the constants.

4 Hypercube

For the hypercube, we show that the ECO model has an advantage over the CO model. Let C_n be an hypercube of 2^n vertices. Each node u is coded by $(x_n, \ldots, x_1)$ with $x_i \in \{0, 1\}$. As for the torus, we assume there exists a global orientation of edges known by each node such that two nodes are adjacent along the direction i, labelled $\rightarrow_i$, if they agree in all but the position i. For $n \geqslant 2$, C_n is hamiltonian (see [6]) and it follows that any subgraph of C_n isomorphic to $C_{n'}$ with $n' < n$ is hamiltonian. Moreover, there exists in C_n an hamiltonian circuit.

In this model, the co-ordination works in the following way : each node always reports first the direction 1, then direction 2, ... , direction n. In other words, if the advice of a node $u = (x_n, \ldots, x_1)$ is $\rightarrow_i$, it indicates that the destination v should have at least $i - 1$ identical coordinates $x_{i-1} \ldots x_1$.

Let us consider the starting node $u = (x_n, \ldots, x_1)$ and the node $v = (y_n, \ldots, y_1)$ is the node containing s. The idea of the algorithm to find the coordinates of v is the following : let $i = 1$; (1) we choose a subgraph $Q' = C_{\lceil 2k+1 \rceil} \in C_n$ such that all nodes of Q' have same coordinates $x_i, y_{i-1}, \ldots, y_1$ (2) we follow an hamiltonian path in Q' and compute, for the first $2k + 1$ nodes, the number m of responses $\rightarrow_i$ of Q'; (3) if $m > k$ then $y_i = 1 - x_i$ else $y_i = x_i$; (4) repeat Step 1 until the item is found.

[2] it may happen that we found s in Step 2 but the walk in the rectangle R is a spiral to obtain the same result

[3] This can be done with $O(k)$ extra steps

Theorem 9. *In an hypercube C_n of 2^n vertices with k liars, a mobile agent of $O(n + \log k)$ bits of memory can find an item in at most $d(2k + 1)$ steps.*

In the ECO Model, a node u gives a response $Q_u = (a_{n-1}, \ldots, a_0)$. The position of s is given using the majority among $2k + 1$. Immediately, an easy upper bound of $d + 4k + 2$ steps can be obtained by following $2k + 1$ nodes in a hamiltonian path in C_n. This result can be improved if we consider only a hamiltonian path in a subgraph of C_n isomorphic to $C_{\lceil \log(2k+1) \rceil}$.

Theorem 10. *In a hypercube C_n of 2^n vertices with k liars, a mobile agent can find an item in at most $d + 2k + 1 + \lceil \log(2k + 1) \rceil$ steps with $O(n \log k)$ bits of memory.*

5 Trees

We pay particular attention on the CO Model for a tree. Indeed, the shortest path is unique and so, we do not have a question of co-ordination of the nodes. We present one algorithm for bounded degree trees. In this case, the ECO model would lead to the same result as for the CO model.

We suppose that we are starting from a node u_1, considered as a root of the tree. Node u_1 gives an orientation of the edges. Each node, except the root, has $\Delta - 1$ incident edges, corresponding to the directions upward, downward 1, downward 2, etc. and labelled $\uparrow, \downarrow_1, \ldots, \downarrow_{\Delta-1}$. By convention, the edge pointing upward is the edge leading to the root. A node u is a *suspect* if its response is upward and if its parent's response is downward. SearchTree(s, k) works as follow: (1) follow the downward advice until either a suspect u_l or s is found (2) traverse the all subtree rooted in u_l of depth $2k$ and choose to follow the k first edges belonging to the path to leaves with the maximum of downward responses, (3) iterate first step. Analyzing SearchTree, we obtain :

Theorem 11. *In a tree of bounded degree Δ of n vertices and k liars, a mobile agent can find an item in at most $d + O((\Delta - 1)^{2k+1})$ steps with $O(k \log \Delta)$ bits of memory.*

It is the first example where the number of steps is exponential in k. However, the next result indicates that the gap between the upper bound and lower bound is not so large:

Theorem 12. *For $k < \log_{\delta-1} n$, there exists a distribution of k liars in the tree of bounded degree δ with n vertices so that the number of steps required to find s is at least $d + \Omega((\delta - 1)^k)$.*

References

[1] Y. Afek, E. Gafni, and M. Ricklin, Upper and lower bounds for routing schemes in dynamic networks, in: Proc. 30th Symposium on Foundations of Computer Science, (1989), 370–375.

[2] S. Albers and M. Henzinger, Exploring unknown environments, *in Proc. 29th Symposium on Theory of Computing*, (1999), 416–425.

[3] B. Awerbuch, B. Patt-Shamir, and G. Varghese, Self-stabilizing end-to-end communication, *Journal of High Speed Networks* **5** (1996), 365–381.

[4] R.A. Baeza-Yates, J.C. Culberson and G.J.E Rawlins, Searching in the plane, *Information and Computation* **106(2)** (1993), 234–252.

[5] D. Bienstock and P. Seymour, Monotonicity in graph searching, *Journal of Algorithms* **12** (1991), 239–245.

[6] P.J. Cameron. *Topics, Techniques, Algorithms.* Cambridge University Press, 1994.

[7] R. Cole, B. Maggs and R. Sitaraman, Routing on butterfly networks with random faults, *in: Proc. 36th Symposium on Foundations of Computer Science*, (1995), 558–570.

[8] S. Dolev, E. Kranakis, D. Krizanc and D. Peleg, Bubbles: Adaptive routing scheme for high-speed networks, *SIAM Journal on Computing*, to appear.

[9] E. Kranakis and D Krizanc, Searching with uncertainty, *in: Proc. 6th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, (1999), C. Gavoille, J.-C, Bermond, and A. Raspaud, eds., pp, 194-203, Carleton Scientific, 1999.

[10] L.M. Kirousis, E. Kranakis, D. Krizanc, and Y.C. Stamatiou. Locating information with uncertainty in fully interconnected networks. unpublished paper, (1999).

[11] E. Kushilevitz and Y. Mansour, Computation in noisy radio networks, *in Proc. 9th Symposium on Discrete Algorithms*, 1998, 236–243.

[12] L. Kirousis and C. Papadimitriou, Interval graphs and searching, *Discrete Mathematics* **55** (1985), 181–184.

[13] N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriou, The complexity of searching a graph, *Journal of the ACM* **35** (1988), 18–44.

[14] T. Leighton and B. Maggs, Expanders might be practical, *in: 30th Proc. Symposium on Foundations of Computer Science*, (1989), 384–389.

[15] P. Panaite and A. Pelc, Exploring unknown undirected graphs, *in: Proc. 9th Symposium on Discrete Algorithms*, (1998), 316–322.

[16] Mobile Agents, W. R. Cockayne and M. Zyda, editors, Manning Publications Co., Greenwitch, Connecticut, 1997.
http://www.manning.com/Cockayne/Contents.html

Complete Exchange Algorithms for Meshes and Tori Using a Systematic Approach

Luis Díaz de Cerio, Miguel Valero-García and Antonio González

Computer Architecture Department - Universitat Politècnica de Catalunya (Spain)[1]

Abstract. Frequently, algorithms for a given multicomputer architecture cannot be used (or are not efficient) for a different architecture. The proposed method allows the systematic design of complete exchange algorithms for meshes and tori and it can be extended to some other architectures that may be interesting in the future.

1 Introduction

Complete exchange is a global collective communication operation in which every process sends a unique block of data to every other process in the system. Since complete exchange arises in many important problems, its efficient implementation on current parallel machines is an important research issue.

Multidimensional meshes and tori have received a lot of attention as convenient topologies for interconnecting the nodes of message-passing multicomputers. The fixed-degree property of these topologies makes them very suitable for scalable systems and solving communication intensive problems, such as complete exchange, become a challenge.

Some authors have proposed algorithms for a given scenario that cannot be applied to other scenarios with a reasonable efficiency. It is also frequent that the ideas which inspire a concrete algorithm for a given architecture cannot be used to derive algorithms for others (i.e. the idea of building spanning graphs, which has inspired efficient algorithms for tori, cannot be applied to meshes, where nodes are not symmetric).

The proposed method enables the systematic design of complete exchange algorithms for a wide range of scenarios, including k-port C-dimensional meshes and tori. The method produces efficient algorithms, outperforming in many cases the best known algorithms for a significant range of the system parameters (number of nodes, problem size, communication parameters, etc.).

We have developed analytical models upon a small set of system parameters. These simple models enable a quick and general comparison among different algorithms and are good enough to show the potential goodness of the method.

1. *Author's address:* Computer Architecture Department, Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08034 Barcelona (Spain). *E-mai:* ldiaz@ac.upc.es

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 591-594, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Considered Scenarios

An scenario is defined by the following features: topology, dimensionality, switching model, port model.

As interconnection topology we consider C-dimensional meshes and tori. The dimensionality of the mesh or torus (C) will take normally the values 1, 2 or 3.

The most frequent switching model is circuit switched. This term includes direct connect, wormhole and virtual cut-through [6]. We model the cost of sending the message in a circuit switched model assuming no conflicts in the use of the system resources, as $t_s + dt_p + Nt_e$, where t_s is the communication start-up, t_p is the time to switch an intermediate node and t_e is the communication time per size unit. A key issue in the design of parallel algorithms for a circuit switched model is the use of conflict-free communication patterns.

The port model defines the number of channels connecting every processor with its local router. In one-port model, every node can send and/or receive only one message at the same time. In all-port model the node can send and/or receive at the same time a message through every external link.

3 The Method

The method (complete description in [2]) starts from a particular parallel algorithm for complete exchange. This algorithm belongs to the CC-cube class of algorithms [1]. CC-cube algorithms are simple and useful to solve many important problems. The original CC-cube algorithm is first transformed in a systematic way through the communication pipelining technique [3], to produce a pipelined CC-cube algorithm. The objective of this transformation is to introduce a certain amount of process level parallelism, which will later enable the exploitation of the machine resources.

The first issue in the mapping of the pipelined CC-cube onto the target scenario is the allocation of the pipelined CC-cube processes to the mesh/torus nodes. The allocation problem can be formulated as an embedding of a hypercube graph onto a mesh/torus graph. We have adopted the standard embedding of hypercubes onto meshes [5] and the xor embedding of hypercubes onto tori [3].

After embedding the pipelined CC-cube onto the mesh/torus, it is necessary to determine how the messages to be exchanged at the end of every iteration of the algorithm must be routed to avoid conflicts. This is the step where the particularities of the target scenario such as circuit switching model and port model play an essential role. The proposed routing algorithms are optimal in terms of communication cost and use a simple dimension-ordered minimal routing policy.

4 A CC-Cube Algorithm for Complete Exchange

In this section we describe the CC-cube algorithm for complete exchange which is used as starting point for the proposed method. Figure 1 shows a particular example in which $d = 3$.

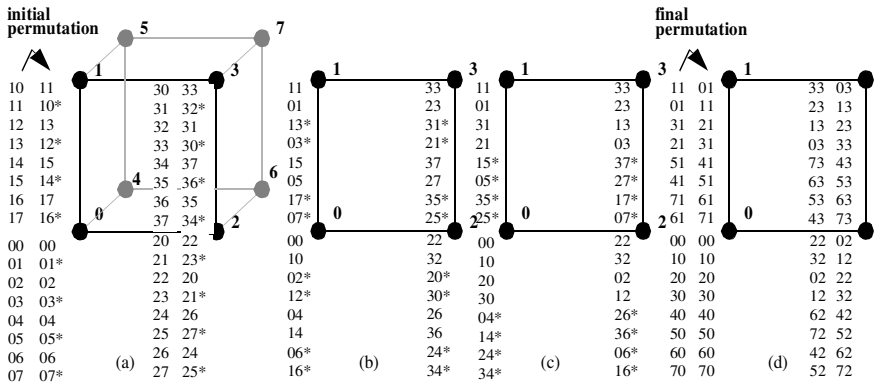


Figure 1 A CC-cube algorithm for complete exchange, with d=3.

Every node initially stores 2^d blocks of data in a vector of blocks M. Each block of data will be identified by the pair (m, j), where m is the source node and j is the destination node for the corresponding block. For clarity, figure 1 only shows the blocks of data corresponding to nodes 0, 1, 2 and 3 of the CC-cube. Note that block (m, j) is initially stored in position $M[j]$.

Initially, every node performs a permutation of vector M. After this permutation, block (m,j) is stored in position $M[m \text{ XOR } j]$, in node m (see figure 1.a). The objective of this permutation is to store a block in the location of M given by the binary value obtained by setting all the bits corresponding to the dimensions that the block must traverse (i.e., $M[3]$ of node 1 stores the block (1,2) since this block must traverse dimensions 0 and 1 to reach its destination). Then, in every iteration i of the CC-cube, a subset of 2^{d-1} blocks are extracted from M to build the vector x_i that will be sent to the neighbor in dimension i. In figure 1, the blocks which are selected in every iteration are marked with an asterisk. Because of the initial permutation, all the nodes obtain their corresponding blocks from the same locations of M. In particular, in iteration i a block in position $M[j]$ is selected if the i-th bit of the binary form of index j is set. After the message exchange, the received blocks are stored in the positions of M which were occupied by the sent blocks.

After the three iterations required for complete exchange, a new permutation is required to leave the blocks in their final positions in M. This permutation is exactly the same as the initial one (see figure 1.d).

The above algorithm was proposed in [4] in order to minimize the maximum orbit length. We propose a slight modification of the algorithm in order to meet the requirements of the communication pipelining technique. This modification refers to the order in which the blocks are sent in each iteration. In particular, to build a message x_i, the blocks are always arranged in reverse order with regard to their positions in M. For instance, x_0 in node 0 contains blocks 07, 05, 03 and 01, in this order.

5 Concluding Remarks

In this paper (a strongly reduced version of [2]), a method for the systematic design of complete exchange algorithms for a wide range of scenarios has been proposed. Starting from a particular algorithm for complete exchange, the method uses communication pipelining to introduce node level parallelism, efficient embeddings of hypercube onto mesh/torus to map the pipelined algorithm onto the target scenario, and an efficient message routing to exploit the communication resources of the machine.

Besides its generality, an important feature of the method is the possibility of tuning the algorithm for the particular machine configuration, by choosing an optimal value of the pipelining degree, which is an algorithm parameter. This feature makes possible to obtain a high performance for a wide range of values of the machine and the problem parameters.

Under common analytical modeling assumption, the algorithms obtained through the proposed method outperform previous proposals for a significant range of values for the machine parameters and block size, in almost all the considered scenarios. In many cases the proposed algorithms are about twice as fast as the best previous proposal.

Acknowledgments

This work has been supported by the Ministry of Education and Science of Spain (CICYT TIC-98/0511) and the European Center for Parallelism in Barcelona (CEPBA).

References

[1] L. Díaz de Cerio, A. González and M. Valero-García, Communication Pipelining in Hypercubes, *Parallel Processing Letters*, Vol.6, No.4, December 1996.

[2] L. Díaz de Cerio, M. Valero-García and A. González, A Systematic Approach to Develop Efficient Complete Exchange Algorithms for Meshes and Tori, *Research Report* UPC-DAC-97-29, http://www.ac.upc.es/recerca/reports/INDEX1997DAC.html

[3] A. González, M. Valero-García and L. Díaz de Cerio, Executing Algorithms with Hypercube Topology on Torus Multicomputers, *IEEE Trans. on Parallel and Distributed Systems,* Vol. 6, No. 8, August 1995, pp. 803-814.

[4] S.L. Jonhnsson and C.-T. Ho, Optimal All-to-All Personalized Communication with Minimum Span on Boolean Cubes, in proceedings of the*6th Distributed Memory Computing Conf.*, 1991, pp. 299-304.

[5] S. Matic, Emulation of Hypercube Architecture on Nearest-Neighbor Mesh-Connectec Processing Elements, *IEEE Trans. on Computers*, vol. 39, No. 5, May 1990, pp. 698-700.

[6] J.G. Peters and M. Syska, Circuit-Switched Broadcasting in Torus Networks, *IEEE Trans. on Parallel and Distributed Systems,* Vol. 7, No. 3, March 1996, pp. 246-255.

Algorithms for Routing AGVs on a Mesh Topology

Ling Qiu and Wen-Jing Hsu

Centre for Advanced Information Systems, School of Applied Science
Nanyang Technological University, Singapore 639798
{P146077466, hsu}@ntu.edu.sg

Abstract: This paper proposes to adapt parallel sorting/message routing algorithms to route Automated Guided Vehicles (or AGVs for short) on mesh-like path topologies. Our strategy guarantees that no conflicts will occur among AGVs when moving towards their destinations; a high degree of concurrency can be achieved during our routing process.

1. Introduction

From a computer science perspective, *Automated Guided Vehicle*s (or AGVs for short) systems are intrinsically parallel and distributed systems that require a high degree of concurrency. Many algorithms for scheduling and routing of AGVs have been proposed, however, most of them are applicable to systems with a small number of AGVs, offering only low degree of concurrency [1]. With drastically increased number of AGVs in recent applications (e.g. in the order of a hundred in a container terminal), efficient scheduling and routing algorithms are needed to resolve the increased contention of resources (e.g. path, loading and unloading buffers) among AGVs. Because they often employ regular route topologies, the new applications also demand innovative strategies to increase system performance.

In this paper, we apply ideas arising from parallel processing and adapt sorting algorithms to route AGVs concurrently on a mesh path. Based on our routing strategy, all AGVs can reach their destinations within $O(n \log n)$ steps of well-defined physical moves in an $n \times n$ mesh. No conflicts, congestion, livelocks or deadlocks among the AGVs will occur, and a very high degree of concurrency can be achieved.

The remaining part of the paper is organized as follows. We describe the problem in Section 2. Section 3 gives the routing algorithm. Section 4 concludes the paper.

2. The Problem

Many AGV applications employ regular path topologies, such as mesh. As a case in point, presently at Nanyang Technological University, Singapore, the application of AGVs in a container handling system is being studied [1 – 4]. The main goal is to schedule and route AGVs within a mesh-like path and container stacking areas (as shown in Fig. 1). In a container port, an AGV could originate from a location near one of the container cranes at a ship, and have a destination at a yard area; similarly, an AGV could also reverse the direction of its trip, i.e., start from a yard location and

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 595-599, 2000.
© Springer-Verlag Berlin Heidelberg 2000

end up near a crane (refer to Fig. 1). It is also possible for an AGV to move from a yard area to another. Based on this reality, we formulate the problem as shown in Fig. 2 which is practical and commonly encountered: AGVs are assumed to originate from arbitrary buffers and supposed to end up at the others, how to efficiently route all AGVs such that they can reach their destinations without conflicts? However, to clearly explain the main ideas of our routing process, we will start from an essential scenario as shown in Fig. 3. In the model, all AGVs start from the intersection of the pathways (referred to as "nodes" subsequently) and end their journeys also at the nodes of the mesh. We will first demonstrate how to adapt the principle of *bitonic sorting* to this model, and then extend the result to our primitive problem.

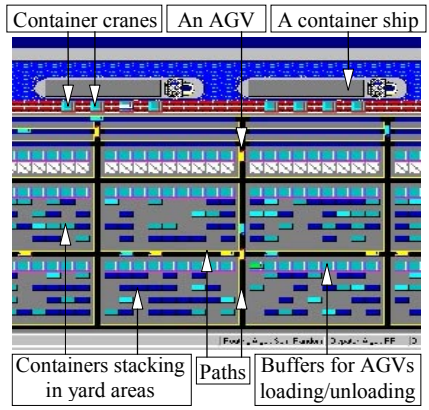


Figure 1. The topology of a container port

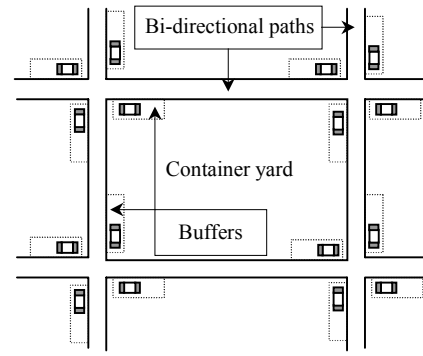


Figure 2. The problem formulation

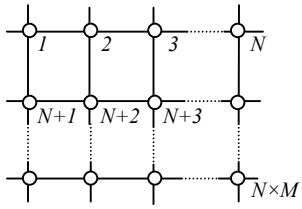


Figure 3. The essential model

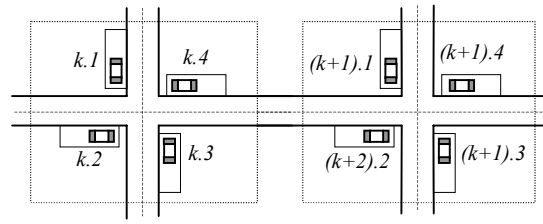


Figure 4. Extended nodes and the numbering of buffers

3. The Routing Strategy

3.1 Routing among Nodes

Referring to Fig. 3, assume that the path topology considered is a mesh of N columns by M ($M = 2^k$) rows. Thus it has $M \times N$ nodes in total, which are numbered from *1* to $M \times N$ in a row fashion. We have $M \times N$ AGVs initially stationed at each node. Every AGV has a unique destination (different from the others). With the numbering of nodes, it should be clear that the routing of the AGVs amounts to sorting their

destination IDs, with the data exchanges corresponding to the moves of AGVs. The major difference is, here we must ensure that the physical moves of AGVs are free of hazards like collisions; the moves may also be constrained in terms of the spatial constraints. The following four steps give a detailed description of the routing algorithm. Readers are referred to [2] for a detailed illustration of the algorithm.

Step 1: Route AGVs in every row concurrently, so that for odd rows AGVs are sorted into increasing order, and for even rows decreasing.

Step 2: Apply *bitonic merging* to route AGVs on row $4i-3$ to row $4i$, where $i = 1, 2, \ldots, M/4$. Finally, AGVs on row $4i-3$ and row $4i-2$ are sorted into increasing order, while AGVs on row $4i-1$ and row $4i$ decreasing order.

In this phase, vehicles first move vertically between row $4i-3$ and row $4i-2$ or row $4i-1$ and row $4i$, then move horizontally within each row.

Step 3: Apply *bitonic merging* to route AGVs on row $8i-7$ to row $8i$, where $i = 1, 2, \ldots, M/8$, we get an increasing sequence of AGVs from row $8i-7$ to row $8i-4$ and a decreasing one from row $8i-3$ to row $8i$.

Step 4: Apply operations similar to *Step 3* to route AGVs on row $2^j i-(2^j-1)$ to row $2^j i$ repeatedly, for $j = 4, 5, \ldots, k$ and $i=1, 2, \ldots, 2^{k-j}$ where $M = 2^k$. After this step, every AGV gets to its final destination.

3.2 Routing among Extended Nodes

Now return to the primitive problem as shown in Fig. 2, in which four AGVs initially station at the buffers near a node. In order to apply the previous routing algorithm, we define an *extended node* as a node together with the four associated buffers nearby (cf. the dotted rectangles as shown in Fig. 4). Every buffer is also numbered in the form of $x.y$, where x is the ID of the node while y is the ID of the buffer in the extended node (cf. Fig. 4). We also assume that every pathway consists of two bi-directional lanes, and that every AGV has distinct destination from the other. If we define that (x.y > u.v) holds iif (x > u) or (x = u and y > v), our routing algorithm can be applied directly to handle this case without any revision. The only difference is that now the number of AGVs is four times as much as the previous one.

Claim 1: Routed by our algorithm, every AGV can reach it destination after limited steps of physical moves. ■

Claim 2: Applying our routing algorithm, no conflicts, congestion or deadlocks will arise amongst AGVs during the routing process. ■

Readers are referred to [2 – 4] for the detailed proofs of both claims.

3.3 Complexity of Concurrent Moves

Now let us analyze the time requirement for AGVs to reach their destinations. For this purpose, assume that the vertical and horizontal path segments are of the *same* length. Moreover, mechanically speaking, making a 90^0-turn (to change from horizontal direction to vertical direction or vice versa) is usually more expensive (in terms of the time, space and energy required) than moving in straight lines. Therefore, we will also analyze the number of 90^0-turns needed in the algorithm.

Definition: A *rectilinear step* is the move required for an AGV to move from a buffer in an extended node to another buffer in an adjacent extended node, exclusive of all 90^0-turns.

Claim 3: Applying our routing algorithm to the scenario shown in Fig. 4, the total amount of all concurrent rectilinear steps for all AGVs to reach their destinations is upper-bounded by $O(M + kN)$ or $O(Max\{M, N\, logM\})$. ■

Theorem 1: Given arbitrary initial configurations as on an $n \times n$ mesh as described in Fig. 2 and Fig. 4, all AGVs will be able to reach their destinations using $O(n\, log\, n)$ concurrent rectilinear steps and $O(log\, n)$ concurrent 90^0-turns. ■

Readers are referred to [2, 4] for the detailed proofs of the claim and the theorem.

4 Discussions & Conclusions

This paper has proposed to adapt parallel sorting algorithm to route AGVs on a mesh path topology. Routed by the routing algorithm, all AGVs can reach their destinations without conflicts and deadlocks. A high degree of concurrency is also achieved.

In Section 3, we assume that there are two bi-directional lanes in a path. However, even if there is only one lane, the routing algorithm still works. The only difference is that in this case every process of swapping has to be finished in two phases, each of which allows AGVs to move in one direction. Hence the total number of concurrent steps of moves is doubled, but the complexity order remains the same. Usually, the cost of space resource is much lower than that of an AGV that can cost as much as a million dollars per vehicle. From this perspective, it is worthy of obtaining a higher system efficiency and AGV utilization by dedicating space for lanes or buffers. The trade-off among space, cost and time (measured by steps of physical moves) is discussed in detail in [2, 4]. Moreover, if the number of AGVs in an extended node is less than four or in other words, some of the buffers do not have AGVs initially, we regard the vacancies as *virtual* AGVs. Virtual AGVs are numbered with the maximum destination IDs in the extended node. Thus algorithm still works.

Before each step of routing process is carried out, the system has the idea of global traffic information upon which the routing decision is made. From this perspective, if we use the centralized control mechanism, every AGV simply follows the instructions sent by the central controller. Whereas under the decentralized control mechanism, the central controller decides when AGVs begin to compare and swap; while the local controllers of AGVs communicate with each other and coordinate their moves.

For further study, one direction is to relax on the constraints of the problem, in which case, a proper scheduling may be required. For instance, we need scheduling when the destinations of AGVs are not all distinct; similarly, when AGVs have continuous tasks, how to schedule and route them [1]? These outstanding problems have obvious applications.

5. References

1 Qiu, L. and Hsu, W.-J., Scheduling and Routing Algorithms for AGVs: a Survey. *Technical report: CAIS-TR-99-26, Centre for Adv. Info. Sys., Schl. of Applied Science., Nanyang Tech. Univ., Singapore, Oct 1999.* Available at http://www.cais.ntu.edu.sg:8000/.

2 Qiu, L. and Hsu, W.-J., Adapting Sorting Algorithms for Routing AGVs on a Mesh-like Path Topology. *Tech. report: CAIS-TR-00-28, Centre for Adv. Info. Sys., Schl. of Applied Sci., Nanyang Tech. Univ., S'pore, Feb 2000*. Available at http://www.cais.ntu.edu.sg:8000/.

3 Qiu, L. and Hsu, W.-J., Conflict-free AGV Routing in a Bi-directional Path Layout. *Proc. of 5th Int'l Conf. on Comput. Integrated Manu., Singapore, Mar 2000, pp. 392-403.*

4 Qiu, L. and Hsu, W.-J., Routing AGVs by Sorting. Accepted for *2000 Int'l Conf. on Para. and Dist. Processing Tech. and App. (PDPTA'2000), Las Vegas, USA, Jun 26-29, 2000.*

Self-Stabilizing Protocol for Shortest Path Tree for Multi-cast Routing in Mobile Networks*

Sandeep K.S. Gupta[1], Abdelmadjid Bouabdallah[2], and Pradip K Srimani[1,2]

[1] Department of Computer Science, Colorado State University, Ft. Collins, CO 80523 USA
[2] Universite de Technologie de Compiegne, Lab. Heudiasic, UMR CNRS 6599, Dep. Genie Informatique, BP 20529, 60205 Compiegne Cedex, France

Abstract. Our objective is to view the topology change as a change in the node adjacency information at one or more nodes and utilize the tools of self-stabilization to converge to a stable global state in the new network graph. We illustrate the concept by designing a new efficient distributed algorithm for multi-cast in a mobile network that can accommodate any change in the network topology due to node mobility.

Keywords: Self-stabilizing Protocol, Distributed System, Multi-cast Protocol, Fault Tolerance, Convergence, System Graph.

1 Introduction

Most of the protocols for designing near optimal multi-cast trees for given multi-cast groups in mobile ad hoc networks and analyzing their performance [AB96, CB94]. assume that the underlying network topology does not change. Recently we have proposed a self-stabilizing protocol for maintaining multi-cast tree in mobile ad hoc network which is based on pruning a minimum weight spanning tree [GS99]. This protocol minimizes the bandwidth requirement for multi-casting a message. In order to minimize the multi-cast latency a shortest-path tree can be employed. A shortest path tree rooted at node r is spanning tree such that for any vertex v, the distance between r and v in the tree is the same as the shortest-path distance in the entire graph. Our purpose in this short note is to show how a self-stabilizing algorithm for shortest path tree generation can be simply adapted to solve the problem of maintaining a hortest-path multi-cast tree in a radio network for a given multi-cast group. We analyze the time complexity of the algorithm in terms of number of rounds needed for the algorithm to stabilize after a topology change, where a round is defined as a period of time in which each node in the system receives beacon messages from all its neighbors.

* Address for Correspondence: Pradip K Srimani, Department of Computer Science, Colorado State University, Ft. Collins, CO 80523, Tel: (970) 491-7097, Fax: (970) 491-2466, Email: srimani@CS.ColoState.Edu

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 600–604, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Shortest Path Tree Protocol

We make the following assumptions about the system. (1) A data link layer protocol at each node i maintains the identities of its neighbors in some list $neighbors(i)$. This data link protocol also resolves any contention for the shared medium by supporting logical links between neighbors and ensures that a message sent over a correct (or functioning) logical link is correctly received by the node at the other end of that link. (2) Each node periodically (at intervals of t_b) broadcasts a *beacon* message. This forms the basis of *neighbor discovery protocol*. When a node i receives the beacon signal from a node j which is not in its neighbors list $neighbors(i)$, it adds j to its neighbors list (data structure $neighbors_i$ at node i), thus establishing link (i, j). For each link (i, j), node i maintains a timer t_{ij} for each of its neighbors j. If node i does not receive a beacon signal from neighbor j in time t_b, it assumes that link (i, j) is no longer available and removes j from its neighbor set. Upon receiving a beacon signal from neighbor j, node i resets its appropriate timer. (3) The topology of the ad-hoc network is modeled by a (undirected) graph $G = (V, E)$, where V is the set of nodes and E is the set of links between neighboring nodes. We assume that the links between two adjacent nodes are always bidirectional. Since the nodes are mobile, the network topology changes with time. We assume that no node leaves the system and no new node joins the system. (4) There is an underlying unicast routing protocol to send unicast messages between two arbitrary nodes in the network.

Each node $i \in V$ maintains a local variable $D_i(r)$; $D_i(r)$ is the current estimate of $S_i(r)$ known at node i and it determines the *local state* of node i. In addition, each node i also maintains a predecessor pointer P_i pointing to one of the nodes in $Adj(i)$; P_i points to the node adjacent to node i in the currently estimated shortest path from node i to node r. The set $\mathcal{N}(i)$ contains neighboring nodes of i that are on currently estimated shortest paths from node i to r. Each node i executes the following code:

$$\textbf{if } (i = r \wedge (D_i \neq 0 \wedge P_i \neq NULL)) \textbf{ then } D_i = 0 \;\&\; P_i = NULL$$

$$\textbf{else if } (i \neq r \wedge (D_i(r) \neq \min_{\forall j \in Adj(i)} (D_j(r) + w_{ij}) \vee P_i \notin \mathcal{N}(i))$$

$$\textbf{then } D_i(r) = \min_{\forall j \in Adj(i)} (D_j(r) + w_{ij}) \;\&\; P_i = k, k \in \mathcal{N}(i)$$

2.1 Complexity Analysis

In case of a mobile ad hoc network, where the SPST protocol is used to maintain a multi-cast routing tree, it is required that the protocol converges as quickly as possible and it is also true that the participating mobile clients (nodes) do not act in a adversarial way i.e., they make their moves according to some known uniform protocol (i.e., each node sends its state to its neighbors at regular intervals). The purpose of this section is to provide an analysis of the convergence time of the proposed protocol under the

assumptions of the ad hoc network model. Each node periodically broadcasts a beacon message to its neighbors and this period is same for each node in the system. Let us define a *round* of computation as the time between two consecutive beacon message broadcast (i.e. the period of beacon message broadcast). Thus, in each round, every node that is privileged due to the actions taken by the nodes during the immediate past round, will make a move to make the node locally stable. Let $\mathcal{D}$ denote the diameter of the underlying network with uniform edge weights (i.e. when each edge is assigned an uniform weight of 1); and m_1 and m_2 denote respectively the minimum and the maximum edge weight in the system graph.

Remark 1. The ratio $\lceil \frac{m_2}{m_1} \rceil$ plays a very important role in determining the convergence time of the protocol. In an adversarial oracle this ratio can be very large (ratio of the largest real number to the smallest positive non zero real number that can be stored), while in the context of an ad hoc network, the ratio (range of link costs) would be small. For example in "revised ARPANET routing metric" the most expensive link is only seven times the cost of the least expensive link. The reason being that there is a relationship link cost and link utilization. A link which has a very low cost gets overly utilized since it is included in many shortest path whereas a link with a very high cost has an extremely low utilization since it hardly gets included in any path. Analysis of Internet packet traces show that, if the range of link cost is very wide, say 1 to 127, a high percentage of network traffic is destined for a small number of network links. This results is overall very poor utilization of the network. In mobile ad hoc network this can aggravate the problem of bandwidth scarcity even further. Hence, in mobile ad hoc networks the ratio of link of most expensive to least expensive link is expected to be very small.

Lemma 1. *Starting from a given illegitimate state consider the system state after p rounds; each of the nodes that are yet to be stabilized has $D_i(r) \geq pm_1$.*

Lemma 2. *Consider a node i which is p hops away from root r (the shortest path from i to r may involve more edges); node i will be stabilized in at most $p\lceil \frac{m_2}{m_1} \rceil$ rounds, after the node r is stabilized.*

Lemma 3. *The upper bound on the number of rounds needed by the entire network to stabilize starting from an arbitrary illegitimate state is given by $\mathcal{D} * \lceil \frac{m_2}{m_1} \rceil + 1$.*

3 Multi-cast Protocol

Our protocol for fault tolerant maintenance of the multi-cast tree for a given source node (we call it root node r) and its multi-cast group consists of 2 logical steps: (1) construction of the shortest path spanning tree of the mobile network graph in presence

of the topology change due to node mobility (establishing unique parent pointer for each node in the SPST); (2) pruning from the SPST the nodes that are not needed to send the message to the multi-cast group members. The protocol described in the previous section maintains the shortest path spanning tree in a fault tolerant way (that accommodates the topology change due to node mobility) as well as maintains the knowledge of the tree in a distributed way; each node knows its unique parent pointer). In this section we describe the protocol to prune the SPST and build the multi-cast tree.

The multi-cast source node r needs to send the message to the members of the arbitrary multi-cast group. Each node in the network knows whether it is a member of the multi-cast group (IS_Member_i is true). Note that even if a node is not a member of the multi-cast group, it will need to transmit the message to its successors iff any of its successors belong to the multi-cast group. In the rooted SPST, each node i can determine if it is a leaf node in the SPST (it has no neighbor node j such that $P_j = i$; in this case, node i will set $Flag_i$ variable to 1 if IS_Member_i is true and to 0 otherwise. Any other node i (i is not a leaf node in the SPST) will look at all its successors in the SPST and will set its $Flag_i$ to 1 iff at least one of its successors either has a Flag of 1 or is a member of the multi-cast group or node i is a member of the the multi-cast group. After this process stabilizes, each node i, when it receives the multi-cast message from its parent in the tree, knows that it needs to forward the message to its successors if $Flag_i$ is 1. Note that the nodes with $Flag_i$ value 1 constitute the multi-cast tree (although not all the nodes in the multi-cast tree are necessarily members of the multi-cast group). Now we can state the complete protocol to maintain the multi-cast tree:

$$\textbf{SPST}\begin{cases} \textbf{if } (i = r \wedge (D_i \neq 0 \vee P_i \neq NULL)) \textbf{ then } D_i = 0 \;\&\; P_i = NULL \\ \textbf{else if } (i \neq r \wedge (D_i(r) \neq \min_{\forall j \in Adj(i)}(D_j(r) + w_{ij}) \vee P_i \notin \mathcal{N}(i)) \\ \qquad \textbf{then } D_i(r) = \min_{\forall j \in Adj(i)}(D_j(r) + w_{ij}) \;\&\; P_i = k, k \in \mathcal{N}(i) \end{cases}$$

$$\textbf{Multi-cast Tree}\begin{cases} \textbf{if } Flag_i \neq \forall_{k \in Adj(i)}((P_k = i) \wedge (IS\_Member_k \vee Flag_k)) \\ \quad \textbf{then } Flag_i = \vee_{k \in Adj(i)}((P_k = i) \wedge (IS\_Member_k \vee Flag_k)) \end{cases}$$

Lemma 4. *Starting from any illegitimate state the protocol correctly sets the Flag for each node which is a member of the multi-cast group, in at most $n-1$ rounds after the SPST protocol has stabilized. Starting from any illegitimate state, the entire protocol stabilizes to a valid multi-cast tree in at most $\mathcal{D} * \lceil \frac{m_2}{m_1} \rceil + n$ rounds.*

References

[AB96] A. Acharya and B. R. Badrinath. A framework for delivering multicast messages in networks with mobile hosts. *ACM/Baltzer Journal of Mobile Networks and Applications*, 1:199–219, 1996.

[CB94] K. Chao and K. P. Birman. A group communication approach for mobile communication. In *Proc. Mobile Computing Workshop*, Santa Cruz, CA, December 1994.

[GS99] S. K. S. Gupta and P. K. Srimani. Using self-stabilization to design adaptive multicast protocol for mobile ad hoc networks. In *Proc. DIMACS Workshop on Mobile Networks and Computing*, Rutgers University, NJ, March 1999.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*. Plenum Press, New York, 1972.

Quorum-Based Replication in Asynchronous Crash-Recovery Distributed Systems

Luís Rodrigues[1] and Michel Raynal[2]

[1] Universidade de Lisboa
ler@di.fc.ul.pt,
[2] IRISA
raynal@irisa.fr

Abstract. This paper describes a solution to the replica management problem in asynchronous distributed systems in which processes can crash and recover. Our solution is based on an Atomic Broadcast primitive which, in turn, is based on an underlying Consensus algorithm. The proposed technique makes a bridge between established results on Weighted Voting and recent results on the Consensus problem.

1 Introduction

Replication is a well known technique to increase the reliability and availability of data [8]. Replication involves coordination among replicas. For instance, replicas may need to agree on a common state after a given action or on the order by which requests will be processed. Several of these coordination activities are instances of the Consensus problem, that can be defined in the following way: each process proposes an initial value to the others, and, despite failures, all correct processes have to agree on a common value (called decision value), which has to be one of the proposed values. Unfortunalety, this problem has no deterministic solution in assynchronous systems where processes may fail, a result known as the Fischer-Lynch-Paterson's (FLP) impossibility result [5].

The FLP impossibility result has motivated researchers to find a set of minimal assumptions that, when satisfied by a distributed system, makes consensus solvable in this system. The concept of unreliable failure detector introduced by Chandra and Toueg constitutes an answer to this challenge [4]. From a practical point of view, an unreliable failure detector can be seen as a set of oracles: each oracle is attached to a process and provides it with information regarding the status of other processes. An oracle can make mistakes, for instance, by not suspecting a failed process or by suspecting a not failed one. The concept has also been extended to the crash-recovery model [1,9,11].

Weighted voting [6] is a well known technique to manage replication in the crash-recovery model. The technique consists in assigning votes to each replica and define quorums for *read* and *write* operations. Quorums for conflicting operations, namely read/write and write/write must overlap such that conflicts can be detected. Typically, voting algorithms are applied in the same context of transactions [7]: quorums ensure one-copy equivalence for each replica, concurrency

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 605–608, 2000.
© Springer-Verlag Berlin Heidelberg 2000

control techniques ensure mutual consistency of data and atomic commitment protocols ensure updates persistence (write operations are performed in the write quorum). It should be noted that, in asynchronous systems, these solutions must also rely on variants of Consensus to decide the outcome of transactions [12].

This paper explores an alternative path to the implementation of quorum-based replication that relies on the use of an *Atomic Broadcast* primitive. An *Atomic Broadcast* primitive allows processes to broadcast and deliver messages in such a way that processes agree not only on the set of messages they deliver but also on the order of message deliveries. By employing this primitive to disseminate updates, all correct copies of a service are delivered the same set of updates in the same order, and consequently the state of the service is kept consistent. The proposed technique makes: *i)* a bridge between established results on Weighted Voting and recent results on the Consensus problem; *ii)* a bridge between the active replication model in the synchronous crash (no-recovery) model and the asynchronous crash-recovery model.

2 System Model and Building Blocks

We consider a system consisting of a finite set of processes. At a given time, a process is either *up* or *down*. When it is *up*, a process progresses at its own speed behaving according to its specification. While being up, a process can fail by crashing: it then stops working and becomes *down*. A down process can later recover: it then becomes up again and restarts by invoking a recovery procedure. The model is augmented with a failure detector so that the Consensus can be solved [9,1,11]. Each process is equipped with two local memories: a volatile memory and a stable storage. When it crashes, a process definitely loses the content of its volatile memory; the content of a stable storage is not affected by crashes.

Processes communicate and synchronize by sending and receiving messages through channels. The quorum-based replica management algorithm requires the use of an unreliable transport protocol and of an atomic broadcast protocol. It has been shown that the atomic broadcast problem is equivalent to Consensus in asynchronous crash-recovery systems [13]. By resourcing to the atomic broadcast protocol our algorithm does not use a Consensus protocol explicitly (the Consensus is encapsulated by the atomic broadcast primitive).

3 Quorum-Based Replica Management

Weighted voting [6] is a popular technique to increase the availability of replicated data in networks subject to node crashes or network partitions. The technique consists in assigning votes to each replica and define quorums for read *read* and *write* operations. Quorums for conflicting operations, namely read/write an write/write must overlap such that conflicts can be detected. Typically, voting algorithms are applied in the context of transactions [7]: quorums ensure one-copy equivalence for each replica, concurrency control techniques ensure mutual

consistency of data and atomic commitment protocols ensure updates persistence (write operations are performed in the write quorum). Here we propose a weighted voting variant based on our atomic broadcast primitive. Votes and quorums are assigned exactly as in the transaction-based weighted-voting algorithms. The atomic broadcast (and the underlying consensus) is defined for the set of data replicas. To maximize availability, the majority condition used in the consensus protocol must be defined using the weights assigned to each replica (this can be achieved with a trivial extension to the protocols of [1,9,11]).

The client of the replicated service does not need to participate in the atomic broadcast protocol. Since the channels are lossy, and processes can crash, the client periodically retransmits its request until a quorum of replies is received. We assume that each client assigns a unique identifier to each request. This identifier is used by the servers to discard duplicate requests and by the client to match replies with the associated request. The read and write procedures simply wait for a read quorum (or write quorum) of replies to be collected. The reply carries the identifier of the request, the data value and version number. The highest version number corresponds to the most recent value of the data, which is returned by the read operation. We avoid locking and keep data available during updates. Thus, reads that are executed concurrently with writes can either read the new or the old data value. To ensure consistency of reads from the same process, each client records the last version read in a variable *timestamp* and discards replies containing older versions. It should be noted that if clients communicate, either directly or by writing/reading other servers, the timestamp must be propagated has discussed in [10].

Each replica keeps the data value and an associate version number. All updates are serialized by the atomic broadcast algorithm. Read operations do not need to be serialized and are executed locally: the quorum mechanism ensures that the client will get the most updated value. Upon reception of a READ request, each replica simply sends a REPLY to the client with its vote, data value, and version number. Upon reception of a WRITE request, the replica first checks if the associated update has already been processed (since the system is asynchronous, the write request can be received *after* the associated update): in such a case, it simply acknowledges the operation. Otherwise, an UPDATE message is created from the write request and atomically broadcast in the group of replicas. Whenever an update is delivered, the value of the data is updated accordingly and the version number incremented. The fact that this update has been applied is logged in the *processed* variable. There is a subtle point regarding the black-box interface between the atomic broadcast protocol and the replication algorithm: when a server recovers it has to parse the sequence of delivered messages, discarding already processed messages.

4 Discussion

Quorum-based techniques to manage replicated data require the write operation to be applied to a number of replicas satisfying a write quorum or applied to none.

When operations are performed in the context of a transaction, a distributed atomic commit protocol [7] is used to decide the outcome of the transaction. Naturally, the atomic commit protocol must be carefully selected to preserve the desired availability, otherwise the execution of this protocol introduces a window of vulnerability in the system. For instance, if a simple two-phase commit protocol is used, the protocol may block even if a replica with a majority of votes remain up [3]. The protocol proposed in this paper shows that weighted voting can also be applied to a strategy that relies on atomic broadcast to manage replicated data in asynchronous crash-recovery systems. An advantage of this approach is that locking is not required during updates. On the other hand, logical clocks are required to ensure consistent reads [10]. The proposed protocol can easily be tailored to implement the Read One replica, Write All strategy. In that case, it encompasses distributed data management protocols based on an atomic broadcast primitive that have been designed in the no-failure model [2].

References

1. M. Aguilera, W. Chen and S. Toueg, Failure Detection and Consensus in the Crash-Recovery Model. *Proc. 12th Int. Symposium on DIStributed Computing (formerly WDAG))*, Andros, Greece, Sringer-Verlar LNCS 1499, pp. 231-245, September 1998.
2. H. Attiya and J. Welch, Sequential Consistency versus Linearizability. *ACM TOCS*, 12(2):91-122, 1994.
3. Ö Babaoğlu and S. Toueg, Understanding Non-Blocking Atomic Commitement. *Chapter 6, Distributed Systems (2nd edition)*, ACM Press (S. Mullender Ed.), New-York, pp. 147-168, 1993.
4. T. Chandra and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2): 225-267, 1996.
5. M. Fischer, N. Lynch and M. Paterson, Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32(2):374-382, 1985.
6. D. Gifford, Weighted Voting For Replicated Data. *Proc. 7th ACM Symposium on Operating Systems Principles*, pp. 150-162, 1979.
7. J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques. *Morgan Kaufmann Pub.*, 1070 pages, 1993.
8. R. Guerraoui and A. Schiper, Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68-74, 1997.
9. M. Hurfin, A. Mostéfaoui and M. Raynal, Consensus in Asynchronous Systems Where Processes Can Crash and Recover. *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette (IN), pp. 280-286, October 1998.
10. R. Ladin, B. Liskov, B. Shrira and S. Ghemawat, Providing Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360-391, 1992.
11. R. Oliveira, R. Guerraoui and A. Schiper, Consensus in the Crash-Recovery Model, *Research report 97-239*, EPFL, Lausanne, Switzerland, 1997.
12. F. Pedone, R. Guerraoui and A. Schiper, Exploiting Atomic Broadcast in Replicated Databases, *Proc. Europar Conference*, Springer-Verlag LNCS 1470, pp.513-520, 1998.
13. L. Rodrigues and M. Raynal. Atomic Broadcast in Asynchronous Crash-Recovery Distributed Systems. *In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pp. 288-295, Taipe, Taiwan, April, 2000.

Timestamping Algorithms: A Characterization and a Few Properties

Giovanna Melideo[1,2], Marco Mechelli[1], Roberto Baldoni[1], and Alberto Marchetti Spaccamela[1]

[1] Dipartimento di Informatica e Sistemistica, Università "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
{Melideo, Mechelli, Baldoni, Marchetti}@dis.uniroma1.it
[2] Dipartimento di Matematica ed Applicazioni, Universitá di L'Aquila,
Via Vetoio, 67100 L'aquila, Italy

Abstract. Timestamping algorithms are used to capture the causal order or the concurrency of events in asynchronous distributed computations. This paper introduces a formal framework on timestamping algorithms, by characterizing some conditions they have to satisfy in order to capture causality. Under the proposed formal framework we derive a few properties about the size of timestamps and local informations at processes obtained by counting the number of distinct causal pasts which could be observed by an omniscient observer during the evolution of a distributed computation.

1 Introduction

Since the Lamport's seminal paper [5], that formalized the notion of causal dependency between events of an asynchronous distributed computation, a lot of work has been carried out to design distributed algorithms that capture the causal dependencies (or the concurrency) between events during a computation [7]. All these algorithms are based on timestamps associated with events and on the piggybacking of information on messages used to update timestamps. If these timestamps represent an isomorphic embedding of the partial order of the computation, the potential causal precedence or the concurrency between two events can be correctly detected just comparing their timestamps, and we say that the algorithm *characterizes causality.*

In this paper we are interested in introducing a formal framework for timestamping algorithms. At this aim, we consider some operational aspects which allow us to characterize some conditions which any timestamping algorithm has to satisfy in order to characterize causality. Under this framework we prove a bijective correspondence among the set $\mathcal{C}_n$ of causal pasts which could be observed during the execution of all distributed computations of n processes, the set $Im_n(\phi)$ of the timestamps which could be assigned by a timestamping algorithm to events in E and the set $Im_n(\mathcal{I}_\ell)$ of local informations maintained by processes.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 609–616, 2000.
© Springer-Verlag Berlin Heidelberg 2000

An interesting result concerns the reckoning of the causal pasts in $\mathcal{C}_n$, which permits to characterize also the cardinality of the sets $Im_n(\phi)$ and $Im_n(\mathcal{I}_\ell)$. This is done by counting all prefix-closed subsets of E with respect to the causal order relation [5] *which can never be causal pasts of any distributed computation.*

By analyzing the size of non-structured information (i.e. the number of bits) necessary to code elements in $Im_n(\phi)$ and in $Im_n(\mathcal{I}_\ell)$ when the timestamping algorithm has to characterize causality, we obtain a confirmation of the Charron-Bost's result [2]. Algorithms which structure timestamps as vectors of k integers characterize causality only if $k \geq n$ (being n the number of processes). Moreover, we also give a property on the minimum size of control information piggybacked on outgoing messages.

These properties partially answer a question of Schwartz-Mattern [8] about the minimum amount of information that has to be managed by a timestamping algorithm which correctly captures the causality.

The remaining of this paper is structured in 5 sections. Section 2 introduces the computation model. Section 3 presents a formal framework for timestamping algorithms. In Section 4 a few properties are given about the size of timestamps, and the amount of information managed by any timestamping algorithm characterizing causality. Finally, Section 5 relates the results obtained in this paper with timestamping algorithms presented in the literature [5,10,4,1].

2 Computation Model

A distributed computation consists of a finite set of n sequential application processes $\{P_1, P_2, \ldots, P_n\}$ which do not share a common memory and communicate solely by message exchanging, with an unpredictable but finite delay. The execution of each process P_i produces a totally ordered set of events E_i. An event may be either *internal* or it may involve communication (*send* or *receive* event).

Let E be *the disjoint union of totally ordered sets* E_i, i.e. $E = \bigcup_{i=1}^{n} E_i$. This set is structured as a partial order by Lamport's causality relation [5], denoted as $\rightarrow$ and defined as follows:

Definition 1. *The causality relation* $\rightarrow \subseteq E \times E$ *is the smallest relation satisfying the following conditions:* $e \rightarrow e'$ *if one of these conditions holds:*
(1) e *and* e' *are events in the same process and* e *precedes* e'
(2) $\exists\ m\ :\ e = send(m)\ \wedge\ e' = receive(m)$
(3) $\exists\ e''\ :\ e \rightarrow e''\ \wedge e'' \rightarrow e'$.

Two events e and e' are concurrent if $\neg(e \rightarrow e')$ and $\neg(e' \rightarrow e)$.

The partial order $\widehat{E} = (E, \rightarrow)$ constitutes a formal model of the distributed computation it is associated with. Namely:

Definition 2. *A relation* $\rightarrow \subseteq E \times E$ *is a causality relation on* E *if [2]:*
(1) $(E, \rightarrow)$ *has no cycles, and*
(2) $\forall e \in E,\ |\{(e, e') \in \rightarrow |\ e' \in E\} \cup \{(e', e) \in \rightarrow |\ e' \in E\}| \leq 1$
(i.e. for every receipt of a message m*, there is a single sending of* m*).*

According to this model, we denote as $e_{i,j} \in E$ a generic j-th event produced by the process P_i, whose type (internal/send/receive) is defined by the specific causality relation $\rightarrow$ considered on these events[1].

Moreover, we model all distributed computations of n processes as the set $\widehat{\mathcal{E}}_n = \{(E, \rightarrow) \mid \rightarrow \in \mathcal{R}_{\rightarrow}\}$, where $\mathcal{R}_{\rightarrow} \subseteq 2^{E \times E}$ denotes the set of all causality relations on E.

For a given computation $\widehat{E} \in \widehat{\mathcal{E}}_n$, the *causal past* of e in $\widehat{E}$ is the prefix-closed set of E under the causal order $\uparrow (e, \widehat{E}) = \{e' \in E \mid e' \rightarrow e\} \cup \{e\}$. Each causal past $\uparrow (e, \widehat{E}) \subseteq E$ can be decomposed in n disjoint subsets $\uparrow_1 (e, \widehat{E}), \uparrow_2 (e, \widehat{E}), \ldots, \uparrow_n (e, \widehat{E})$, where $\uparrow_i (e, \widehat{E}) = \uparrow (e, \widehat{E}) \cap E_i$.

Following Schwartz-Mattern [8], $\forall\ \widehat{E} \in \widehat{\mathcal{E}}_n$, $(\{\uparrow (e, \widehat{E}) \mid e \in E\}, \subseteq)$ is an isomorphic embedding of $(E, \rightarrow)$. In fact, different causal pasts in the same computation correspond to different events, and

$$\forall\ \widehat{E} \in \widehat{\mathcal{E}}_n,\ \forall e, e' \in \widehat{E}\ (e \neq e'), \quad e \rightarrow e' \Leftrightarrow \uparrow (e, \widehat{E}) \subset \uparrow (e', \widehat{E}). \tag{1}$$

We denote as $\mathcal{C}_n = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}_n} \{\uparrow (e, \widehat{E}) \mid e \in E\}$ the set of causal pasts which could be observed during the execution of all computations of n processes.

3 A Characterization of Timestamping Algorithms

Techniques to detect causality relations or concurrency between events are based on timestamps of events produced by the execution of a timestamping algorithm, which assigns "on-the-fly", that is during the evolution of the computation $\widehat{E}$ and without knowing its future, to each event e a value $\phi(e, \widehat{E})$ of a suitable partially ordered set $(D, <)$.

A *timestamping algorithm* $\mathcal{A}$ is usually characterized by a partially ordered set $(D, <)$ called *timestamps domain*, a *timestamping function* ϕ which establishes a correspondence between events of a computation and timestamps in D, and a set of rules implementing the algorithm which decide both local information at processes and control information piggybacked by messages.

The aim is to assign values in D to events so that for each $\widehat{E} \in \widehat{\mathcal{E}}_n$, the suborder $(\{\phi(e, \widehat{E}) \mid e \in E\}, <)$ of the timestamps assigned to events is an isomorphic embedding of $(E, \rightarrow)$.

This is usually formalized [2,3,8] by requiring the function ϕ *characterizes causality*, i.e. ϕ is injective and $\forall \widehat{E} \in \widehat{\mathcal{E}}_n, \forall e, e' \in \widehat{E},\ e \rightarrow e' \Leftrightarrow \phi(e, \widehat{E}) < \phi(e', \widehat{E})$.

We denote as $Im_n(\phi) = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}_n} \{\phi(e, \widehat{E}) \mid e \in E\} \subseteq D$ the set of timestamps which could be assigned by any timestamping algorithm to events during the execution of all computations of n processes.

[1] When referring to generic events we drop subscripts and we use the following simple notation e, e' and e''.

A bijective correspondence between $Im_n(\phi)$ and $Im_n(\mathcal{I}_\ell)$. A deterministic timestamping algorithm assigns a timestamp to an event e only basing on the current local control information at the process producing e. So, it will assign the same timestamp to two events which have the same local information at process, even though they belong to two distinct distributed computations.

We denote as $\mathcal{I}_\ell(e, \widehat{E})$ the local control information associated with e by the timestamping algorithm during the execution of the computation[2]. Namely, $\mathcal{I}_\ell$ is a function mapping events to values in an ordered set $(L, \prec)$, called *local domain*. Then, we can say that

$$\forall\ \widehat{E}_1, \widehat{E}_2 \in \widehat{\mathcal{E}}_n,\ \mathcal{I}_\ell(e, \widehat{E}_1) = \mathcal{I}_\ell(e', \widehat{E}_2) \Rightarrow \phi(e, \widehat{E}_1) = \phi(e', \widehat{E}_2). \tag{2}$$

Let $Im_n(\mathcal{I}_\ell) = \bigcup_{\widehat{E}\in\widehat{\mathcal{E}}_n} \{\mathcal{I}_\ell(e, \widehat{E}) \mid e \in E\}$ be the set of local informations which could be associated with events by any timestamping algorithm, during the execution of all computations of n processes. The following proposition shows that any timestamping algorithm which characterizes causality is characterized by a bijective correspondence between $Im_n(\mathcal{I}_\ell)$ and $Im_n(\phi)$. In fact, it proves that if the timestamping function characterizes causality then the converse of (2) is also true.

Proposition 1. *If a timestamping function characterizes causality, then*

$$\forall\ \widehat{E}_1, \widehat{E}_2 \in \widehat{\mathcal{E}}_n,\ \mathcal{I}_\ell(e, \widehat{E}_1) = \mathcal{I}_\ell(e', \widehat{E}_2) \Leftrightarrow \phi(e, \widehat{E}_1) = \phi(e', \widehat{E}_2). \tag{3}$$

Proof. Sufficiency is given by equation (2). To prove necessity, let $e \in \widehat{E}_1,\ e' \in \widehat{E}_2$ be two events with the same timestamp (i.e. $\phi(e, \widehat{E}_1) = \phi(e', \widehat{E}_2)$) and $\mathcal{I}(e, \widehat{E}_1) \neq \mathcal{I}(e', \widehat{E}_2)$. Since a timestamping algorithm cannot predict the progress of the computation, there could exist an event $e'' \in \widehat{E}_1$ such that $\mathcal{I}(e'', \widehat{E}_1) = \mathcal{I}(e', \widehat{E}_2)$. In this case, condition (2) implies that $\phi(e'', \widehat{E}_1) = \phi(e', \widehat{E}_2)$, that is the algorithm must assign to event e'' the same timestamp as e'. By hypothesis $\phi(e, \widehat{E}_1) = \phi(e', \widehat{E}_2)$, so $\phi(e, \widehat{E}_1) = \phi(e'', \widehat{E}_1)$ holds, that is in the same computation two events have the same timestamp. As ϕ characterizes causality, $\phi(e, \widehat{E})$ and $\phi(e'', \widehat{E})$ must be distinct.

A bijective correspondence between $\mathcal{C}_n$ and $Im_n(\phi)$. If ϕ characterizes causality, the condition (1) implies that $\forall \widehat{E} \in \widehat{\mathcal{E}}_n$, $(\{\phi(e, \widehat{E}) \mid e \in E\}, <)$ is an isomorphic embedding of $(\{\uparrow (e, \widehat{E}) \mid e \in E\}, \subseteq)$, i.e.

$$\forall \widehat{E} \in \widehat{\mathcal{E}}_n,\ \forall e, e' \in \widehat{E}, (e \neq e'),\ \uparrow (e, \widehat{E}) \subseteq \uparrow (e', \widehat{E}) \Leftrightarrow \phi(e, \widehat{E}) < \phi(e', \widehat{E}). \tag{4}$$

We consider an omniscient observer whose role is to instantaneously detect if a pair of events is causally related or concurrent only by comparing their timestamps.

[2] We suppose there is no redundant local information at processes, i.e. the local information is minimal.

The condition (1) implies *the observer must have perfect knowledge of all causal pasts at any time*, so it can be argued that the timestamps known by the observer $(Im_n(\phi), <)$ have to form an isomorphic embedding of $(\mathcal{C}_n, \subseteq)$. This implies the *decoding function* $\varphi : Im_n(\phi) \rightarrow \mathcal{C}_n$, which characterizes the algorithm executing by the observer, is *bijective* and satisfies the following condition: $\forall d_1, d_2 \in Im_n(\phi),\ d_1 < d_2 \Leftrightarrow \varphi(d_1) \subseteq \varphi(d_2)$.

Previous condition directly implies (4). Moreover, since φ is bijective and $\phi = \varphi^{-1} \circ \uparrow$, we can assert that causal pasts and timestamping functions characterizing causality are also related as follows:

$$\uparrow (e, \widehat{E}_1) = \uparrow (e', \widehat{E}_2) \Leftrightarrow \phi(e, \widehat{E}_1) = \phi(e', \widehat{E}_2). \tag{5}$$

The operational aspects of timestamping algorithms analyzed in the previous paragraphs allow us to argue that both $Im_n(\phi) \subseteq D$ and $Im_n(\mathcal{I}_\ell) \subseteq L$ are actually a coding of the set $\mathcal{C}_n$. Then, we can characterize a timestamping algorithm as a sequence $\mathcal{A}(\overline{D}, \overline{L}, \chi_D, \chi_L)$ where:

- $\overline{D} = (D, <)$ is a partial order called *timestamps domain*;
- $\overline{L} = (L, \prec)$ is a partial order called *local domain*;
- $\chi_D : \mathcal{C}_n \rightarrow D$ is a mapping from causal pasts to timestamps;
- $\chi_L : \mathcal{C}_n \rightarrow L$ is a mapping from causal pasts to local informations.

Definition 3. *A timestamping algorithm* $\mathcal{A}(\overline{D}, \overline{L}, \chi_D, \chi_L)$ characterizes causality *if (i)* χ_D *and* χ_L *are both injective functions, and (ii) the function* $\phi = \chi_D \circ \uparrow$ *characterizes causality (i.e.* $\mathcal{A}$ *characterizes causality if* ϕ *characterizes causality and it timestamps events according to (3) and (5)).*

An Example of Timestamping Algorithm: Vector Clocks [3,6]. The Vector Clocks algorithm codifies causal pasts as integer vectors of size n. Let VC_i be the vector clock endowed by the process P_i. $VC_i[j]$ represents the number of events on P_j in the causal past known by P_i. In this case:

(1) $\overline{D} \equiv \overline{L} \equiv (I\!N^n, \leq)$, where $\forall\ V, V' \in I\!N^n,\ V \leq V'$ iff $\forall i,\ V[i] \leq V'[i]$;
(2) $\chi_D \equiv \chi_L : \mathcal{C}_n \rightarrow I\!N^n$ is defined as: $\forall i,\ \forall S \in \mathcal{C}_n,\ \chi_D(S)[i] = |S \cap E_i|$.

4 Causal Pasts of a Set of Events E

In this section we provide some interesting properties on the set of causal pasts $\mathcal{C}_n$. Moreover, being $|Im_n(\phi)| = |Im_n(\mathcal{I}_\ell)| = |\mathcal{C}_n|$, we are interested in the reckoning of elements in $\mathcal{C}_n$, obtained as a corollary of Propositions 2 and 3.

The following proposition gives necessary conditions so that a prefix-closed subset of events $S \subseteq E$ could be a causal past. We recall that each causal past $S \in \mathcal{C}_n$ can be decomposed in n subsets $S_1, S_2, \ldots, S_n$, where $S_i = S \cap E_i$.

Proposition 2. *Let* $S \subseteq E$ *be a prefix-closed subset of events generated by* n *processes.* S *is a causal past* ($S \in \mathcal{C}_n$) *only if* $S \neq \emptyset$ *and when the number* k *of nonempty subsets in its decomposition is at least* 3, $|S| \geq 2(k-1)$.

Proof. The first claim easily follows by the definition of causal past, because at least e belongs to $\uparrow (e, \widehat{E})$, so $S \neq \emptyset$. If k processes have events in the causal past S, then at least $k-1$ processes have to send messages in order to establish a dependency. Each of $k-1$ messages contributes 2 events to S. Hence, $2(k-1)$ is the minimum number of events in S when $k \geq 3$.

The previous proposition proved that if k subsets, with $3 \leq k \leq n$, are nonempty in the decomposition of a prefix-closed subset S and $|S| \leq 2k-3$, then $S \notin \mathcal{C}$ In the following proposition we count the number of these sets, in order to obtain a precise reckoning of $\mathcal{C}_n$.

Proposition 3. *The number of prefix-closed subsets $S \subseteq E$ of size at most $2k-3$ which can be decomposed in $k \geq 3$ nonempty subsets is:*

$$\sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}. \tag{6}$$

Proof. By applying basic mathematical enumeration results, since (i) k nonempty subsets can be on any of n processes and (ii) the number of prefix-closed sets S of size h which can be decomposed in k *nonempty* subsets is $\binom{h-1}{h-k}$, we have that the number required is: $\sum_{k=3}^{n} \binom{n}{k} \sum_{h=k}^{2k-3} \binom{h-1}{h-k} = \sum_{k=3}^{n} \binom{n}{k} \sum_{h=0}^{k-3} \binom{h+k-1}{h}$.

The value $\binom{h+k-1}{h}$, denoted as $N(h,k)$, represents the number of prefix-closed subsets of size h which can be decomposed in *at most* k subsets. It can be easily proved that $N(h,k) = \sum_{i=0}^{h} N(i, k-1)$, so the thesis (6) follows by $\sum_{h=0}^{k-3} \binom{h+k-1}{h} = \sum_{h=0}^{k-3} N(h,k) = N(k-3, k+1) = \binom{2k-3}{k}$.

For simplicity's sake and wlog we assume processes generate m event each.

Corollary 1. *If n processes generate m events each, then*

$$|\mathcal{C}_n| = (m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}.$$

Proof. If each process generates m events, we have $(m+1)^n$ different prefix-closed subsets of events. The thesis follows by considering that $\emptyset \notin \mathcal{C}$and there are $\sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}$ prefix-closed subsets which cannot be causal pasts (Eq. 6).

4.1 Properties

The Corollary 1 and the Properties 1 and 2 directly imply that:

Property 1. A timestamping algorithm characterizes causality only if $|Im_n(\phi)| = |Im_n(\mathcal{I}_\ell)| = (m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}$.

As a consequence, the coding of each element in $Im_n(\phi)$ and $Im_n(\mathcal{I}_\ell)$ requires at least $\lceil \log_2 |\mathcal{C}_n| \rceil = \lceil \log_2 ((m+1)^n - 1 - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k}) \rceil$ bits. Regarding local information at processes, from an operational point of view, the empty set

is usually used in the initial step, so in practice it is necessary to locally use at least $\lceil \log_2 (|Im_n(\phi)| + 1) \rceil = \lceil \log_2 ((m+1)^n - \sum_{k=3}^{n} \binom{n}{k} \binom{2k-3}{k})) \rceil$ bits.

Property 2 gives the necessary amount of information piggybacked on messages, when the timestamping algorithm characterizes causality locally maintaining only minimal control informations, that is codings of causal pasts.

Let $\mathcal{I}_p(send(m), \widehat{E})$ be the control information piggybacked upon message m, and $Im_n(\mathcal{I}_p) = \bigcup_{\widehat{E} \in \widehat{\mathcal{E}}_n} \{\mathcal{I}_p(e, \widehat{E}) \mid e \in E\}$ be the set of control informations which could be piggybacked upon messages during the execution of all distributed computations on n processes (if e is not a send event we assume $\mathcal{I}_p(e, \widehat{E}) = \emptyset$).

Property 2. A timestamping algorithm characterizes causality only if $|Im_n(\mathcal{I}_p)| = (m+1)^{n-1} - 1 - \sum_{k=3}^{n-1} \binom{n-1}{k} \binom{2k-3}{k}$.

Proof. Let $e_{u,h}$ be a send event and $e_{i,j}$ the corresponding receive event in any computation $\widehat{E}$. By definition, $\uparrow (e_{i,j}, \widehat{E}) = \uparrow (e_{i,j-1}, \widehat{E}) \cup \uparrow (e_{u,h}, \widehat{E}) \cup \{e_{i,j}\}$. If we denote $S_k = \uparrow_k (e_{i,j-1}, \widehat{E})) \cup \uparrow_k (e_{u,h}, \widehat{E})$, we have $\uparrow (e_{i,j}, \widehat{E}) = \uparrow_i (e_{i,j}, \widehat{E}) \cup (\bigcup_{k \neq i} S_k)$. Then, distinct values of $\uparrow (e_{i,j}, \widehat{E})$ are associated to different values of $\bigcup_{k \neq i} S_k$, which are as many as all possible causal pasts which involve events in $n-1$ processes. Consequently their number is $(m+1)^{n-1} - 1 - \sum_{k=3}^{n-1} \binom{n-1}{k} \binom{2k-3}{k}$.

As a consequence the coding of each element in $Im_n(\mathcal{I}_p)$ requires at least $\lceil \log_2 ((m+1)^{n-1} - 1 - \sum_{k=3}^{n-1} \binom{n-1}{k} \binom{2k-3}{k})) \rceil$ bits.

A remark on the Vector Clock algorithm. If n processes generate m events each, $D = L = \{0, \ldots, m\}^n$, so $|D| = |L| = (m+1)^n$. By Proposition 2, D and L are redundant. In fact, $D, L \supset Im_n(\phi) = Im_n(\mathcal{I}_\ell) = \{V \in \{0, \ldots, m\}^n \mid k_V \geq 3 \Rightarrow \sum_{i=1}^{n} V[i] \geq 2(k_V - 1)\}$, where k_V denotes the number of indices i such that $V[i] /= 0$, implying $|D|, |L| > \mathcal{C}_n$. Namely, D and L include vectors (as [1,1,1]) which can never be associated with events, codifying prefix-closed subsets which are not causal pasts.

To codify all elements in D and L it is necessary to use at least $n \lceil \log_2 (m+1) \rceil$ bits, that is more than necessary information. However, from an operational point of view, a coding that excludes non-potential causal pasts seems to be not practicable. As a consequence, a timestamping algorithm based on vector clocks provides the closest coding to the minimal quantity of information required.

5 Related Work

Previous properties give a theoretical confirmation to the fact that some timestamping algorithms such as *scalar clock* [5], *plausible clocks* [10] and *direct dependency vectors* [4] are not able to characterize causality on-the-fly[3].

[3] A deep discussion about these timestamping algorithms is out of the scope of this paper. Nice surveys can be found in [7,8].

Plausible clocks maintain locally at each process a vector of $k < n$ entries, that is less than the quantity required by property 1. The *scalar clocks* are a particular case of plausible clocks when considering $k = 1$.

A timestamping algorithm based on *direct dependency* tracking meets the requirement of property 1 as each process maintains locally a vector of integers of size n. However, each message piggybacks only one integer, that is the index of the send event of the sender process, so by Property 2, it is not appropriate to characterize causality. On the other hand, it is well-known that the timestamping algorithm based on *direct dependency* can off-line (i.e., with some additional computation) reconstruct all causality relations between events [8]. This give rise to an interesting remark: *if a timestamping algorithm satisfies Property 1 but not Property 2, it has the necessary information for characterizing the causality relation but it needs some extra (off-line) computation.*

Previous observation is the baseline of the *k-dependency vector* algorithm introduced in [1], where, given an integer $k \leq n$, each process piggybacks a subset of size k of the local vector, including the current index of the send event of the sender process (as in direct dependency algorithm) and other $k - 1$. The choice of the other $k - 1$ values is left to a scheduling policy of the algorithm.

Acknowledgments
We acknowledge the support of the EU ESPRIT LTR Project "ALCOM-IT" under contract n. 20244.

References

1. R. Baldoni, A., M. Mechelli and G. Melideo. A General Scheme for Dependency Tracking in Distributed Computations, *Technical Report n. 17.99*, Dipartimento di Informatica e Sistemistica, Roma, 1999.
2. B. Charron-Bost. Concerning the size of logical clocks in distributed systems, *Information Processing Letters*, 39, 11–16, 1991.
3. C. Fidge. Timestamps in message passing system that preserve the partial ordering, *Proc. 11th Australian Computer Science Conf.*, 55–66, 1988.
4. J. Fowler and W. Zwaenepoel. Causal distributed breakpoints, *Proc. of 10th IEEE Int'l. Conf. on Distributed Computing Systems*, 134–141, 1990.
5. L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, *Comm. ACM*, **21**7, 558-564, 1978.
6. F. Mattern. Virtual time and global states of distributed systems, M. Cosnard and P. Quinton eds. *Parallel and Distributed Algorithms* 215-226, 1988.
7. M. Raynal and M. Singhal, Logical Time: Capturing Causality in Distributed Systems, *IEEE Computer*, 29(2):49-57, 1996.
8. R. Schwarz and F. Mattern, *Detecting causal relationships in distributed computations: in search of the holy grail*, Distributed Computing 7(3), 149–174, 1994.
9. M. Singhal and A. Kshemkalyani, An Efficient Implementation of Vector Clocks. *Information Processing Letters*, 43:47-52, 1992.
10. F. J. Torres-Rojas and M. Ahamad, Plausible Clocks: constant size logical clocks for distributed systems, *Proceedings of the International Workshop on Distributed Algorithms*, 71–88, 1996.

Topic 10
Programming Languages, Models, and Methods

Paul H.J. Kelly, Sergei Gorlatch, Scott Baden, and Vladimir Getov

Topic Chairmen

1 The Field

This topic provides a forum for the presentation of the latest research results and practical experience in parallel programming. Advances in algorithmic and programming models, design methods, languages, and interfaces are needed for construction of correct, portable parallel software with predictable performance on different parallel and distributed architectures.

Our topic emphasises results which improve the process of developing high-performance programs. Of particular interest these days are novel techniques by which parallel software can be assembled from reusable parallel components without compromising efficiency. Related to this is the need for parallel software to adapt, both to available resources and to the problem being solved.

2 The Common Agenda

The discipline of parallel programming is characterised by its breadth — there is a strong tradition of work which combines

- Programming languages, their compilers and run-time systems
- Formal methods for program specification, derivation and verification
- Architectural issues – both influencing parallel programming, and influenced by ideas from the area – including cost/performance modeling
- The wider techniques for parallelism originated software engineering

This research area has benefited particularly strongly from theoretical ideas. There is a very fruitful tension between, on the one hand, a *reductive approach*: develop tools to deal with program structures and behaviours as they arise, and on the other, a *constructive approach*: design software and hardware in such a way that the optimisation problems which arise are structured, and presumably, therefore, more tractable. To find the right balance, we need to develop theories, languages, cost models, and compilers - and we need to learn from practical experience building high-performance software on real computers.

It is interesting to reflect on the papers presented here, and observe that despite their diversity, this agenda really does underly them all.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 617–619, 2000.
© Springer-Verlag Berlin Heidelberg 2000

3 The Selection Process

We would like to extend our thanks to the authors of the 29 submitted papers, and to the 82 referees who kindly and diligently participated in the selection process.

Eleven papers are presented in full-length form, reflecting a remarkably strong field. Our decision was normally supported by four referees; in three cases only two or three of the four or five referees supported the paper, but, after extensive and enjoyable discussion, the programme committee decided to accept them in order to bring attention to interesting new work.

One of the strengths of Euro-Par is the tradition of accepting new and less mature work in the form of short papers. We were very pleased to select 9 submissions in this category. Brevity is a virtue, and several short papers propose interesting new approaches which we hope to see developed further in time for next year's conference.

4 The Papers

The 20 accepted papers have been assigned to five sessions based on their subject area:

- **Compilation and Performance Issues**
  This session begins with a distinguished paper by Theobald, Kumar, Agrawal, Heber, Thulasiram and Gao, reporting on their implementation of sparse matrix-vector multiply on a simulation of EARTH, their novel multithreaded architecture. The application is a critical kernel for many important applications, and a key motivation behind the design of multithreaded machines. The theme - understanding actual performance issues and the consequences for programming - is continued with work on the predictive value of the BSP cost model, the performance benefit of programming a scalable shared-memory machine using distributed-memory techniques, and the influence of language on the compiler's ability to optimise array algorithms.
- **Structured Parallel Programming**
  This session begins with Zavanella's paper on his optimising compiler for a "skeleton" language which combines data- and task-parallelism. The key idea here is to exploit program structure so that a simple BSP cost model is applicable, and to use this in automatic performance optimisation - thus "tuning" the program for different target machines. The remainder of the session examines this "skeleton" approach from different perspectives - semantics, skeletons and design patterns, skeletons for the computational "grid", and evaluating the performance impact of foreknowledge of the communication pattern - structured parallel programming should lead to "oblivious" BSP programs.

- **Distributed Applications and Java**
  Our third session is devoted to programming distributed applications efficiently and to the particular role of the Java language. R. van Nieuwpoort, Kielmann and Bal describe their distributed-memory implementation of divide-and-conquer, using task stealing and serialisation on demand. The session continues with work on structuring concurrency control in multithreaded programs, and using Java to encapsulate conventional high-performance applications in order to use the language's powerful techniques for coordinating networked computations.
- **Efficient Implementation Techniques**
  This session starts with the paper by Nieplocha, Ju and Straatsma, which describe their implementation of a remote memory access library for a distributed-memory machine with SMP nodes. The next paper concerns the trade-off between concurrency (and potential parallelism) and synchronisation overheads, with the salutary conclusion that cooperative multitasking can be better even on a parallel machine. The last paper of the session asks how well performance portability is achieved in a parallel functional language implementation.
- **Novel Parallel Languages and Formalisms**
  Our final session exhibits several current trends in designing and implementing new languages for parallel programming. It starts with the presentation by Costa, Rocha and Silva, an extended analysis of a tricky implementation problem in parallel logic programming: how to manage environments which are shared and updated by different processes. The remainder of the session deals with a variety of novel parallel languages based mostly on functional formalisms.

The common ground shared by the 20 papers presented here lies in understanding the goals and problems in parallel programming models and languages. What is also very striking is the diversity of approaches being taken!

HPF vs. SAC — A Case Study

Clemens Grelck and Sven-Bodo Scholz

University of Kiel
Dept. of Computer Science and Applied Mathematics
{cg,sbs}@informatik.uni-kiel.de

Abstract. This paper compares the functional programming language SAC to HPF with respect to specificational elegance and runtime performance. A well-known benchmark, red-black SOR, serves as a case study. After presenting the HPF reference implementation alternative SAC implementations are discussed. Eventually, performance figures show the ability to compile highly generic SAC specifications into machine code that outperforms the HPF implementation on a shared memory multiprocessor by a factor of about 3.

1 Introduction

Programming language design basically is about finding the best possible tradeoff between support for high-level program specifications and runtime efficiency. In the context of array processing, data parallel languages are well-suited to meet this goal. Replacing loop nestings by language constructs that operate on entire arrays rather than on single elements, not only improves program specifications; it also creates new optimization opportunities for compilers [3, 4, 1, 8, 7].

FORTRAN-90/HPF introduce a large set of *intrinsics*, built-in operations that manipulate entire arrays in a homogeneous way and that are applicable to arrays of any dimensionality and size. While this allows for concise specifications of many algorithms, code becomes less generic if operations have to be applied to subsets of array elements only. Although regularly structured cases are addressed by the *triple notation*, a step back to loops and scalar specifications often is inevitable. In either case, the resulting code must be tailor-made for a concrete dimensionality. Moreover, FORTRAN-90/HPF also provide no means to build abstractions upon intrinsics other than by sacrificing their general applicability to arrays of any shape.

SAC is a functional C-variant with extended support for arrays [9]. It allows for high-level array processing similar to APL. The basic language construct for specifying array operations is the so-called WITH-*loop*. WITH-loops define `map`- or `fold`-like operations in a way that is invariant to the dimensionalities of argument arrays. As a consequence, almost all operations, typically found as built-in functions in other array languages, can be defined through WITH-loops in SAC without any loss of generality [6]. This concept allows for both: comprehensive array support through easily maintainable libraries and far-reaching customization opportunities for programmers.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 620–624, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In Section 2 we investigate the specificational benefits of SAC in terms of generic high-level programming compared to HPF. In Section 3, we find out how much of a performance penalty has actually to be paid for the increased level of abstraction. Since the SAC-compiler allows to implicitly generate code for shared memory multiprocessors [5], we focus on this architecture. Eventually, Section 4 concludes.

2 A Case Study: The PDE1-Benchmark

As reference implementation for the case study, we chose the PDE1-benchmark as it is supplied by the distribution of the ADAPTOR HPF compiler [2]. PDE1 is a red-black SOR for approximating three-dimensional Poisson equations. The core of the algorithm is a stencil operation on a three-dimensional array u: for each inner element $u_{i,j,k}$, the values of the 6 direct neighbor elements are summed up, added to a fixed number $h^2 f_{i,j,k}$, and subsequently multiplied with a constant factor. Assuming `NX, NY,` and `NZ` to denote the extents of the three-dimensional arrays `U, U1,` and `F`, this operation in the reference implementation is specified as:

```
  U1(2:NX-1,2:NY-1,2:NZ-1) =                                  &
&              FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+           &
&    U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+           &
&    U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+           &
&    U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
```

However, this operation has to be applied to two disjoint sets of elements (the *red* elements and the *black* elements) in two successive steps. This is realized by creating a three-dimensional array of booleans `RED` and embedding the array assignment shown above into a `WHERE` construct.

The given HPF solution can be carried over to SAC almost straightforwardly. Rather than using the triple notation of HPF, in SAC, the computation of the inner elements is specified for a single element at index position `iv`, which by means of a WITH-loop is mapped to all inner elements of an array `u`:

```
u1 = with (. < iv < .) {
       st_sum = u[iv+[1,0,0]] + u[iv-[1,0,0]] + u[iv+[0,1,0]]
              + u[iv-[0,1,0]] + u[iv+[0,0,1]] + u[iv-[0,0,1]];
     } modarray (u, iv, factor * (hsq * f[iv] + st_sum));
```

Note here, that the usage of `<` instead of `<=` on both sides of the generator part restricts the elements to be computed to the inner elements of the array `u`.

The disadvantage of this solution is that it is tailor-made for the given stencil. In the same way the access triples in the HPF-solution have to be adjusted whenever the stencil changes, the offset vectors have to be adjusted in the SAC solution. These adjustments are very error-prone; in particular, if the size of the stencil increases or the dimensionality of the problem has to be changed. To alleviate these problems, we abstract from the problem specific part by introducing an array of weights `W`. In this particular example, `W` is an array of shape `[3,3,3]` with all elements being `0` but the six direct neighbor elements of the center element, which are set to `1`. With such an array `W`, relaxation can be defined as:

```
u1 = with (. < iv < .) {
       block = tile( shape(W), iv-1, u);
     } modarray( u, iv, factor * (hsq * f[iv] + sum( W * block)));
```

In this specification, for each inner element of `u1` a sub-array `block` is taken from `u` which holds all the neighbor elements of `u[iv]`. This is done by applying the library function `tile(` *`shape, offset, array`*`)` which creates an array of shape *shape* whose elements are taken from *array* starting at position *offset*. The computation of the weighted sum of neighbor elements thus turns into `sum( W * block)`, where `(` *`array * array`* `)` refers to an elementwise multiplication of arrays, and `sum(` *`array`*`)` sums up all elements of *array*.

Abstracting from the problem specific stencil data has another advantage: the resulting program does not only support arbitrary stencils but can also be applied to arrays and stencils of other dimensionalities without modifications. Note here, that the usage of `shape(W)` rather than `[3,3,3]` as first argument for `tile` is essential for achieving this.

Although the error-prone indexing operations have been eliminated by the introduction of `W`, the specification still consists of a problem specific WITH-loop which contains an elementwise specification of the relaxation step. It should be mentioned here, that the elementwise specification can be "lifted" into a nesting of operations on entire arrays leading to specifications as they can typically be found in APL programs [6].

After defining relaxation on the entire array, the operation has to be restricted to subsets of the array elements, i.e. to the sets of red and black elements. In the same way as in the HPF program, an array of booleans can be defined which masks the elements of the red set.

For avoiding computational redundancy, the restriction to red/black elements in the HPF solution is realized by integrating it into the relaxation algorithm itself. In SAC, we want to keep these specifications separated in order to improve program modularity as well as its potential for code reuse. Therefore, a shape-invariant general purpose function `CombineMasked( mask, a, b)` is defined, which according to a mask of booleans combines two arrays into a new one:

```
inline double[] CombineMasked( bool[] mask, double[] a, double[] b)
{
  c = with(. <= iv <= .)
      genarray( shape(a), (mask[iv]? a[iv]: b[iv]));
  return( c);
}
```

Provided that `mask`, `a`, and `b` are identically shaped, a new array `c` of the same shape is created, whose elements are copied from those of the array `a` if the mask is true, and from `b` otherwise. Using this function, red-black relaxation can be defined as:

```
u = CombineMasked( red, relax(u, f, hsq), u);
u = CombineMasked( !red, relax(u, f, hsq), u);
```

Note here, that the black set is referred to by `!red`, i.e., by using the elementwise extension of the negation operator (`!`).

3 Performance Comparison

This section presents the essence of thorough investigations on the performance of the HPF- and various alternative SAC-implementations of PDE1 on a 12-processor SUN Ultra Enterprise 4000. The ADAPTOR HPF-compiler v7.0 [2], Sun f77 v5.0, and PVM 3.4.2 for shared memory were used to evaluate the HPF code, the SAC compiler v0.9 and Sun cc v5.0 to compile the SAC code.

single node performance		
runtime	HPF	SAC
64^3	283ms	84ms
256^3	22.2s	6.6s
memory	HPF	SAC
64^3	10MB	8MB
256^3	450MB	260MB

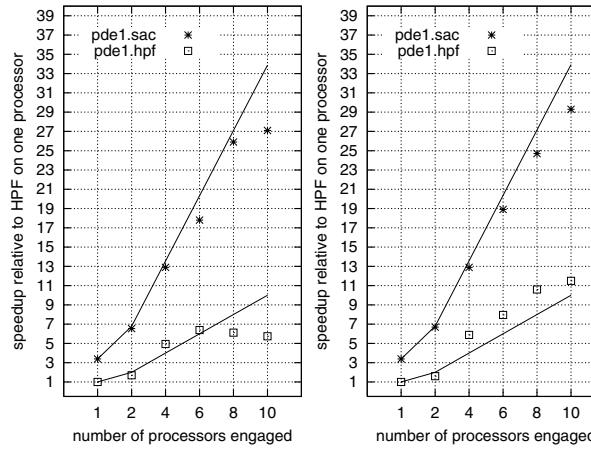


Fig. 1. Runtime performance of SAC and HPF implementations of the PDE1 benchmark, problem sizes 64^3 (center) and 256^3 (right).

One interesting result is that with respect to the accuracy of the timing facility all different SAC specifications — among them those presented in Section 2 — achieve the same runtimes. Having a look into the compiled code reveals that the SAC compiler manages to transform all of them into almost identical intermediate representations. This is mostly due to a SAC-specific optimization technique called WITH-LOOP-FOLDING [10] that aggressively eliminates intermediate arrays. Fig. 1 shows performance results for the problem sizes 64^3 and 256^3. Upon sequential execution, SAC outperforms HPF by a factor of 3.4 for both problem sizes; SAC also needs much less memory: 260MB instead of 450MB in the 256^3 case. This decrease in memory consumption can also be attributed to WITH-LOOP-FOLDING.

Multiprocessor runtimes of the HPF- and SAC-code are shown as speedups relative to HPF single node runtimes. For 64^3 elements, HPF scales well up to 6 processors; any additional processor leads to absolute performance degradation. In contrast, the SAC runtimes scale linearly up to 8 processors and even achieve an additional speedup with 10 processors engaged. The HPF performance scales much better for the problem size 256^3. So, the usage of PVM as low-level communication layer is no principle hindrance to achieve good performance on a shared memory architecture. Nevertheless, even with 10 processors SAC outperforms HPF by a factor of 2.5.

4 Conclusion

The major design goal of SAC is to combine highly generic specifications of array operations with compilation techniques for generating efficiently executable code. By means of a case study, this paper investigates different opportunities for the specification of the PDE1 benchmark in SAC and compares them to the HPF reference implementation in terms of specificational elegance and reusability. Despite their increasingly higher levels of abstraction the various SAC implementations clearly outperform the given HPF program on a shared memory multiprocessor. This shows that high-level generic program specifications and good runtime performance not necessarily exclude each other.

References

[1] G.E. Blelloch, S.Chatterjee, J.C. Hardwick, J. Sipelstein, and M.Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California*, pages 102–111, 1993.

[2] T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, 1994.

[3] D.C. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore, California, 1989.

[4] D.C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[5] C. Grelck. Shared Memory Multiprocessor Support for SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of Implementing Functional Languages (IFL '98), London, Selected Papers*, volume 1595 of *LNCS*, pages 38–54. Springer, 1999.

[6] C. Grelck and S.-B. Scholz. Accelerating APL Programs with SAC. In O. Lefevre, editor, *Proceedings of the Array Processing Language Conference (APL'99), Scranton, Pa.*, volume 29(1) of *APL Quote Quad*, pages 50–57. ACM Press, 1999.

[7] E.C. Lewis, C. Lin, and L. Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. ACM, 1998.

[8] G. Roth and K. Kennedy. Dependence Analysis of Fortran90 Array Syntax. In *Proc. PDPTA'96*, 1996.

[9] S.-B. Scholz. **S***ingle* **A***ssignment* **C** *– Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. PhD thesis, University of Kiel, 1996.

[10] S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.

Developing a Communication Intensive Application on the EARTH Multithreaded Architecture

Kevin B. Theobald[1], Rishi Kumar[1], Gagan Agrawal[2], Gerd Heber[3], Ruppa K. Thulasiram[1], and Guang R. Gao[1]

[1] Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716, USA,
{theobald,kumar,rthulasi,ggao}@capsl.udel.edu,
http://www.capsl.udel.edu

[2] Department of Computer and Information Sciences, University of Delaware,
agrawal@cis.udel.edu,
http://www.cis.udel.edu

[3] Cornell Theory Center, Cornell University, Ithaca, NY 14853, USA,
heber@tc.cornell.edu,
http://www.tc.cornell.edu

Abstract. This paper reports a study of sparse matrix vector multiplication on a parallel distributed memory machine called EARTH, which supports a fine-grain multithreaded program execution model on off-the-shelf processors. Such sparse computations, when parallelized without graph partitioning, have a high communication to computation ratio, and are well known to have limited scalability on traditional distributed memory machines. EARTH offers a number of features which should make it a promising architecture for this class of applications, including local synchronizations, low communication overheads, ability to overlap communication and computation, and low context-switching costs. On the NAS CG benchmark Class A inputs, we achieve linear speedups on the 20-node MANNA platform, and an absolute speedup of 79 on 120 nodes on a simulated extension. The speedup improves to 90 on 120 nodes for Class B. This is achieved without inspector/executor, graph partitioning, or any communication minimization phase, which means that similar results can be expected for adaptive problems.

1 Introduction

One of the most difficult challenges in parallel processing is obtaining high performance for a variety of applications in the presence of high communication and synchronization costs. Multithreaded architectures promise scalable performance for both regular and irregular applications. These systems hide communication and synchronization costs by letting a processor switch to a different thread when a long-latency operation is encountered, and by keeping the cost of this switching low. Multithreaded systems based on dataflow models of computation,

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 625–637, 2000.
© Springer-Verlag Berlin Heidelberg 2000

such as EARTH [1, 2], offer a further benefit of permitting local control between producers and consumers of data rather than expensive global barriers.

This paper presents an important case study to examine the performance of sparse matrix vector multiply (MVM) on EARTH. Sparse MVM is an important and time-consuming kernel in sparse linear algebra problems, including Conjugate Gradient (CG). We have chosen this because it leads to a very high communication to computation ratio, when parallelized without *graph partitioning* [3] and/or an *inspector/executor* paradigm [4], due to the sparseness of the matrix, and does not perform well on conventional distributed memory machines. We show that even without these techniques, a multithreaded system with local synchronization, overlapping of communication and computation, and low-overhead communication and thread-switching can efficiently parallelize sparse MVM.

The goal is to compute a product $q = Av$, where A is an $n \times n$ matrix and v is a vector of length n. Typically, fewer than 1% of the elements of A are non-zero. A common representation for sparse matrices is Compressed Row Storage (CRS), in which only the non-zeroes of a row are stored, and a separate array (colidx) holds their column positions.

The NAS Parallel Benchmark suite [5] includes a version of CG without graph partitioning. Shared memory machines show reasonably good *relative* speedups (absolute speedups are not reported) [5]. For instance, the cache-coherent SGI Origin-2000 has a speedup of 28 on 64 nodes, while the speed of the non-coherent Cray T3E improves by 25 going from 2 to 64 nodes (single-node performance is not reported). The IBM SP-2 distributed memory machine is the most "off-the-shelf," as it has no hardware support for shared memory; its relative speedup is only 13 on 64 nodes. CG results are generally not reported for PC clusters as their relatively slow networks result in terrible speedups.

We have implemented sparse MVM on the EARTH multithreaded system (described in Sect. 2). In our code, the program on each processor is executed as a sequence of threads where the enabling of a thread is *event driven*. Point-to-point, split-phase style communication is performed between processors, generating events to trigger thread execution in a fully asynchronous manner. This, coupled with the low-cost spawning and termination of threads, contributes to highly scalable performance. Execution on each processor is not broken into separate computation and communication phases and global synchronization is not required. Since graph partitioning and inspector/executor are not used, the same approach can be used for parallelization of linear solvers for adaptive problems, i.e., problems where the matrix A is frequently modified.

We report speedups from different problem sizes and different EARTH configurations in this paper. With the NAS Class A sparse matrix (14,000 rows), we observe linear speedups on our 20-node MANNA distributed memory machine [6]. A simulated expansion of the hardware to more nodes yields an *absolute* speedup of 59 on 120 nodes for purely off-the-shelf systems on Class B (75,000 rows), rising to 90 on 120 nodes when a small chip specifically supporting the EARTH execution model is added. A series of experiments reveal the factors leading to high scalability in our implementation.

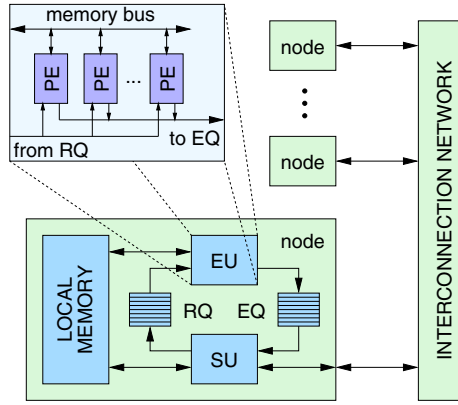


Fig. 1. EARTH Architecture

The rest of the paper is organized as follows. In Sect. 2 we review the details of the EARTH architecture. Our multithreaded approach for sparse MVM is described in Sect. 3. We present our scalability results in Sect. 4, an analysis of how the features of the EARTH system contribute to these results (based on additional experiments) in Sect. 5, and our conclusions in Sect. 6.

2 The EARTH Multithreaded Architecture

EARTH (Efficient Architecture for Running THreads) [1, 2] supports a multithreaded program execution model in which a program is divided into a two-level hierarchy of *fibers* and *threaded procedures*. Fibers are non-preëmptive and are scheduled atomically using dataflow-like synchronization operations initiated by the fibers themselves. These "EARTH operations" make the control and data dependences between fibers explicit, and fibers are scheduled by the rule that one is eligible to begin execution as soon as all relevant dependences have been met. Since fibers can't be interrupted, the producer and consumer of a long-latency operation, such as a data transfer, should be in different fibers.

This model allows the use of local synchronizations between fibers using only relevant dependences, rather than global barriers. It also enables an effective overlapping of communication and computation, by allowing a processor to grab any fiber whose data is ready when an existing fiber terminates after initiating a data transfer.

Conceptually, an EARTH node (see Fig. 1) has an *Execution Unit* (EU), which runs the fibers, and a *Synchronization Unit* (SU), which determines when fibers are ready to run, and handles communication between nodes. There is also a Ready Queue (RQ) of fibers waiting to run on the EU, and an Event Queue (EQ) containing requests for EARTH operations, generated by fibers in the EU.

Because EARTH fibers are non-preëmptive, they are ideal for off-the-shelf processors. This is a big advantage, since the costs of developing and introducing a new processor architecture can be prohibitive. Machines can be developed for the EARTH execution model in an *evolutionary* manner [2]. One can begin with an off-the-shelf parallel machine, and gradually replace its stock components with hardware specially designed to support EARTH operations. In this paper (and previous studies) we consider four possible configurations:

Single: Each node has only one processor, which must alternate between the tasks of the EU and the SU.

Dual: Each node has two processors; one performs the EU tasks and the other emulates the behavior of the SU. The Ready and Event Queues are stored in memory shared by the two processors.

External SU: Each node has a regular off-the-shelf processor and a custom hardware SU. The SU can be built fairly cheaply, yet be optimized for performing the EARTH operations [2, 7]. The EU communicates with the SU through special memory addresses.

Internal SU: This is like the External SU, except that the CPU and SU cores are combined in one package. The interface is the same (memory addresses), but communication between them is off the main bus and hence faster.

3 Multithreaded Implementation

We assume that A is too large to have a complete copy on every node, and needs to be divided among all p nodes. Unless one uses *algorithmic* techniques to reduce communication (see Sect. 1), the simplest way to divide MVM is to split A into p regular strips or blocks. Our algorithm divides A into vertical sections $A_1, \ldots, A_p$. The vector v is also partitioned into sections corresponding to the strips of A. During one multiplication, each node i multiplies its own A_i and v_i, producing a partial result q_i of size n. Neither A nor v have to move, but the vectors $q_1, \ldots, q_p$ must be added to produce the final answer.

The only communication required is the reduction of the components of q. The reference MPI implementation of the NAS CG code adds the q_i vectors using a binary tree. Therefore, the running time of the reduction is $O(n \log p)$.

An alternative is to *pipeline* the reduction in a linear chain. The computation is divided into p phases; the first two are illustrated in Fig. 2 (where $p = 4$). During each phase, node i multiplies one part of its A_i with v_i, producing a part of q_i with only n/p elements. This piece is then sent to the left neighbor (mod p). The starting positions are staggered so that that piece can be added to what the left neighbor produces in the next iteration, as shown in Fig. 2(b).

The total communication burden is the same as for the binary tree. However, here it is evenly balanced among all nodes, so the reduction takes only $O(n)$. Furthermore, by pipelining the reduction, it can be effectively overlapped with the computation (if the architecture allows this).

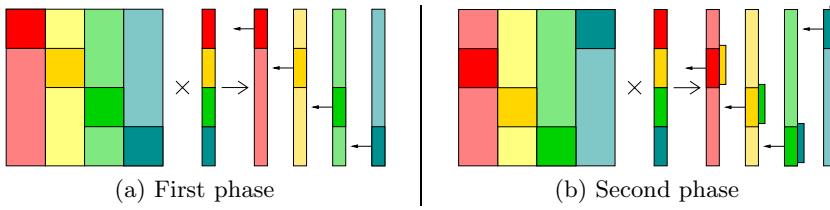


Fig. 2. Pipelining of 4-Node MVM Reduction (First 2 Phases)

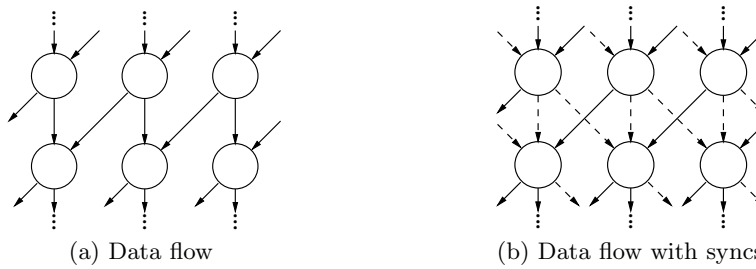


Fig. 3. Implementation of MVM on EARTH

This pipelined reduction is a straightforward specialization of Cannon's algorithm [8] to one dimension. Its implementation in a conventional parallel machine, however, can be challenging because of the frequent communication steps. The high degree of pipelining can hurt performance on conventional coarse-grain parallel machines in at least some of the following ways:

1. If the overheads of sending a message are too high, the overheads become collectively much more significant with p phases than with $\log p$ phases.
2. If a global barrier is required to synchronize the communication of the pieces of q from one phase to the next, then any temporary load imbalance from one phase to the next will force all nodes to wait for the slowest node. (Such variances are likely in a sparse matrix.)
3. If separate communication and computation phases are required, the opportunity to overlap these is lost.

EARTH, on the other hand, is specifically designed for fine-grain synchronization, low-overhead communications, and asynchronous local control. It is therefore an ideal platform for this algorithm. In Sect. 5, we show quantitatively how these properties of EARTH contribute to the performance of MVM.

Fig. 3 shows how the MVM algorithm is transformed to an EARTH program. The computation is broken into a sequence of fibers (a). Each circle represents one fiber, which performs the multiplication of one $n/p \times n/p$ section of A with some v_i. A column of fibers (circles) runs on one node, and represents successive iterations on the same node. Arcs represent data and control dependences. The

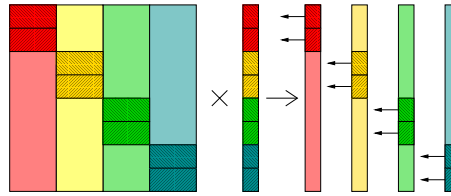


Fig. 4. Multithreading MVM

solid arcs represent the data (in this case, pieces of q_i between nodes), while the dashed arcs represent synchronization signals only. (In this program, all iterations are executed by one program fiber, which is repeatedly instantiated and doesn't need to "send" data to itself.)

On a machine with global barriers, this diagram would adequately describe the simple control structure of the program. However, we want to exploit the features of the EARTH program execution model, namely, multithreading, local synchronization between fibers, and overlapping of communication and computation. We use EARTH's *sync slots* to permit a fiber to start as soon as all required control and data dependences are met, which in this case means 1) the previous iteration on the same node has finished, and 2) the q_i block from the previous iteration has been received from the right neighbor.

However, there is a catch here. If a node always sends its q_i output to the *same* buffer on its left neighbor, the fiber running iteration j must wait until the fiber running iteration j on the left neighbor has finished reading the data from the buffer, or else data could be overwritten. Yet there is no signal path to inform the right neighbor when it is clear to send. Therefore, we add such paths, as shown in part (b). Furthermore, this synchronization can't occur within one iteration, since fibers in EARTH are atomic and non-preëmptive (one can't synchronize "part" of a fiber). There must be a downward movement of sync signals, as shown in (b). Therefore, we allocate *two* buffers in each node, and use one buffer on odd-numbered iterations and the other buffer on even iterations.

This implementation now allows local synchronization between nodes, but doesn't allow overlapping of communication and computation. The fibers in iteration j must wait for the fibers in iteration $j-1$ to finish and send their results. If the architecture has separate hardware for communications, then processors could be sitting idle waiting for the communication to complete. Since EARTH assumes separate communication hardware (a separate CPU or specialized Synchronization Unit), we want to take advantage of this feature.

To do this, we exploit the other major feature of EARTH: multithreading. We split each block multiplication into two halves, each of which produces half of the result vector. This is shown in Fig. 4. Each half is computed by a separate fiber. Now the top halves and the bottom halves of the block multiplications can occur concurrently, as long as each has its own buffers. Essentially, the program in Fig. 3(b) is replicated for each half.

The code studied here uses the CRS format described in Sect. 1, with a separate structure for each A_i. The algorithm was adapted to handle strips of different sizes, so the number of nodes doesn't need to divide n. The code was written in Threaded-C [1, 2], an explicitly threaded programming language, which extends ANSI-C with EARTH operations. Ordinary sequential code was used for the core block multiplication, which dominates the execution time. This routine is a 2-D loop, and our C compiler optimizes the inner loop at the expense of the outer. For the partitioned version, the overhead of the outer loop is much more critical, since the inner loop has far fewer iterations (as explained in Sect. 5). As our compiler lacks any flags or pragmas to favor the outer loop, the core was rewritten in assembly language (43 instructions).[1]

4 Scalability Results

The experiments in this study are based on the EARTH implementations for the MANNA [6]. This machine has 20 dual-processor nodes and can run the Single and Dual configurations listed in Sect. 2. Our experiments were run using SEMi, an accurate, cycle-by-cycle simulator of the MANNA's processors, system bus, memory and interconnection network [1, 2]. The difference in the clock cycle counts between the simulator and the real MANNA have been measured and are typically less than 2% on real benchmarks. We have extended the simulator to model faster processors based on the CPU, bus speed and cache parameters of the PowerPC 620-based PowerMANNA, the MANNA's successor. Here we assume 200MHz processors, with each node having a 200MByte/sec connection to the network. The specialized SU hardware is also simulated, and its speed and interface characteristics are based on the existing MANNA hardware. This gives us confidence that the results obtained from simulating the specialized hardware are reasonably close to what could be achieved with real hardware. As a further check, the MVM code was run on the real MANNA up to 20 nodes for the Single and Dual configurations; the speedup results are nearly identical.

The matrices used come from the NAS Conjugate Gradient (CG) benchmark [5]. We used the Class A (n = 14,000) and B (75,000) problem sizes. These matrices contain 1.9 and 13.7 million non-zeroes, respectively.

Results for the two inputs on the 4 different EARTH configurations are shown in Fig. 5 and 6. The speedups shown are *absolute*, i.e., the parallel performance is compared against the *sequential* code rather than the single-node *parallelized* code. On 120 nodes, the speedups achieved for Class A on the Single, Dual, External SU, and Internal SU versions are 28, 44, 63, and 79, respectively. In the case of Single, the one-processor threaded version is slower than the sequential version by a factor of 68%. For the other three configurations, the degradation of the threaded version on one processor is less than 7%. The Single version has a high overhead of supporting threads, because no extra hardware is available for

[1] An improved native C compiler should make this unnecessary. This kernel is not used for the sequential version as it actually runs slower than the compiled version on large blocks.

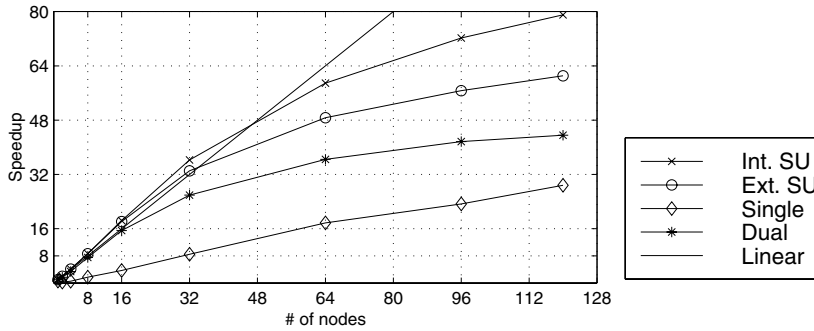


Fig. 5. Speedup on Class A (14,000 Rows)

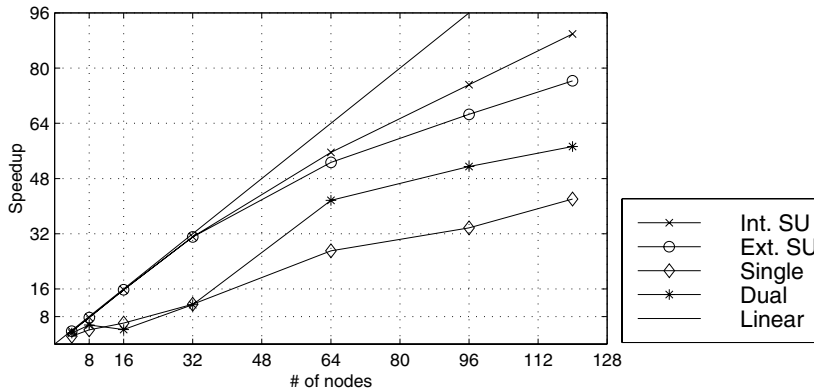


Fig. 6. Speedup on Class B (75,000 Rows)

performing the actions of the SU. We believe that the results for the External and Internal versions on 120 nodes are very encouraging, considering that the problem is not very large for that many nodes. The speedups of the Single, Dual, External SU, and Internal SU versions for Class B on 120 nodes are 44, 59, 78, and 90, respectively.

We have also written a threaded version of the full NAS CG benchmark, which has a number of reduction operations besides the MVM. Although we have not yet conducted a full set of experiments, our initial results show that the CG code has the same scalability as the MVM code. This is mainly because the MVM loop takes more than 95% of the total execution time of CG.

5 Performance Analysis

In this section, we examine the MVM performance in greater detail. We wish to answer two questions:

1. What limits the performance of our implementation of MVM on EARTH?
2. How important are the defining characteristics of the EARTH program execution model to the performance results?

To answer these questions, we ran a series of experiments with Class A input on the same platforms in Sect. 4.

A major loss of efficiency comes from the partitioning itself. The core which multiplies A_i (in whole or in part) with v_i is a 2-D loop, but the sparseness of A limits the iterations of the inner loop when p is large. For instance, the Class A input averages fewer than 140 non-zeroes per row, which means that on 120 nodes, each row of A_i has usually only one or two non-zeroes. The overheads due to partitioning must be significant irrespective of how it is parallelized.

To estimate the upper bound of the performance of our algorithm, we ran a special version of the sequential code, in which we partitioned A and v into p parts and multiplied them part by part, rather than in a single pass. The multiplication of each part was further broken into p stages. This mimics the kind of partitioning seen in the parallel version. On the other hand, to model the beneficial cache effects that often accompany parallelization, this code reuses the same (n/p)-element array for holding partial sums. While this produces incorrect results, it does not affect the *control* structure of the code, and so tells us how much benefit we are likely to get from cache effects.

Thus, for this algorithm, if the modified sequential code runs k times slower than the normal sequential code for a partition factor of p, that suggests the partitioning overheads will limit the speedup on an ideal parallel machine with p nodes to p/k. If cache benefits dominate, then $k < 1$, and thus a superlinear speedup may be attained. In the graphs that follow, we include the curve calculated in this manner as an "Upper bound" speedup.

In Sect. 3, we argued that the multithreading and local synchronization provided by EARTH were essential to getting the most performance from the MVM algorithm. To test this argument, we ran experiments on two modified versions of the Threaded-C MVM code, in which features of EARTH are removed. This way we can measure the benefits quantitatively.

The first experiment ("No multithreading") removes the overlapping of communication and computation by having a single fiber per iteration on each node, as in Fig. 3(b). This code has the local synchronization feature of EARTH but does not take advantage of the ability to switch to another thread of execution during a long-latency operation such as transferring a block of data.

The second experiment ("Global barrier") removes the benefits of local synchronization from the preceding experiment by simulating the effect of a global barrier in the no-multithreading code. In this experiment, SEMi halts each node when it is about to begin or end a communication phase, and when all nodes

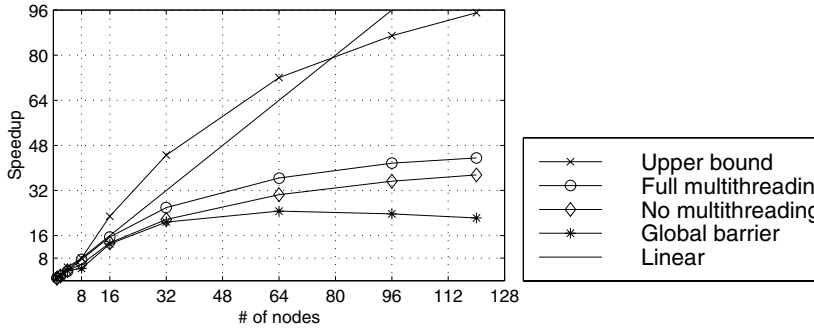


Fig. 7. Comparison of Parallel Versions on Dual

have halted, SEMi causes all nodes to continue where they halted. (The synchronizations are still local, but the earlier nodes are forced to wait for the last node.) The "global barrier" thus simulated is instantaneous, and all we are measuring is the cost of having some nodes wait for others, not the cost of the synchronization itself.

Results for the Dual and Internal SU configurations are shown in Fig. 7–8.[2] Each curve shows the original speedup curve from Sect. 4 ("Full multithreading"), the theoretical upper bound, and the speedups for the two simulations described above. We can make three main observations:

1. When we have efficient hardware support for EARTH's communication and multithreading operations, the speedup curves of our fully multithreaded implementation are reasonably close to the "upper limit" according to the limitations of our partitioning strategy. This tells us that our Threaded-C implementation is very effective at exploiting the parallelism which is inherent in the algorithm.
2. For this application, local synchronization gives a great improvement in performance. Global synchronization eliminates the ability to tolerate temporary imbalances in the load among the nodes. We observed that our uniform partitioning balanced the static work per node to within 10% of average,[3] and that over time, the load on one node stays *roughly* in sync with the other nodes [9].[4] However, there can be slight variations from iteration to iteration. While these variations average out in the long run, they can cause a slowdown if a global barrier always forces the node with less work to wait

[2] Other data can be found in our technical report [9].

[3] If p is not a divisor of n, then the last node will have fewer columns than the others. Uniform balancing may not occur with other types of inputs. However, since the current program is already able to handle different numbers of columns on each node, it would be easy to adjust the sizes of the A_i strips by counting non-zeroes.

[4] The staggered starting position is important, since most sparse matrices are far denser near the main diagonal.

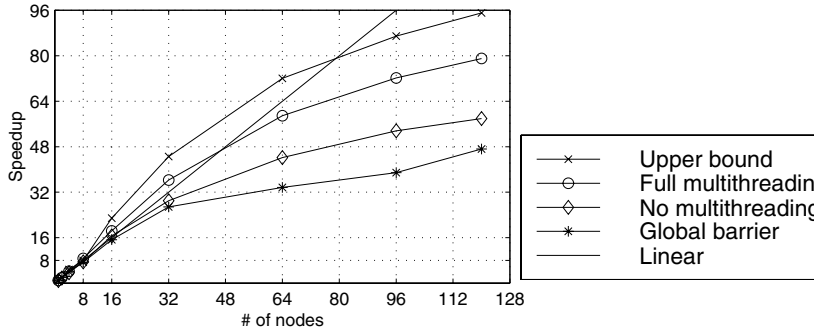


Fig. 8. Comparison of Parallel Versions on Internal SU

for other nodes to catch up. EARTH's local dataflow synchronization mechanism allows for looser coupling between nodes.

3. Finally, the ability to divide code into multiple threads of control is helpful in overlapping computation and communication. When we exploit this ability in our Threaded-C code, the performance improves roughly 15%.

Additional statistics collected by SEMi showed that the network is almost 40% saturated with the multithreaded code on Class A, showing that we make effective use of the communication hardware.

6 Conclusion

The aim of this study was to implement the sparse MVM core computation part of the CG linear systems solver on the EARTH multithreaded system, and to evaluate and analyze its performance. Sparse MVM has typically not performed well on conventional distributed memory machines, due to the high communication costs. As mentioned in Sect. 3, a straightforward multiplication program needs to communicate $O(n)$ data between each pair of neighboring nodes. Therefore, it is important to overlap this communication with computation and to minimize all other overheads wherever possible.

We chose a straightforward partitioning strategy for our implementation, and optimized it to take advantage of the special features of the EARTH multithreading model. Fine-grain threads ("fibers") in our code are synchronized strictly according to which data they need rather than through global barriers. Partitioning the program into multiple fibers permits overlapping of computation with communications. Low overheads in performing the multithreading and communication operations reduce the costs of frequent data transfers.

The current implementation of MVM on EARTH was shown to hide the communication latency effectively, thereby increasing the performance to a very high level. For example, a speedup of 59 is attained with 120 processors (on Class

B) when strictly off-the-shelf processors are used. Specialized hardware support for the EARTH model can increase the speedup to 90 on 120 nodes, without compromising the use of off-the-shelf processors for the main CPU.

The program was written in Threaded-C, an explicitly threaded variant of C which makes the features of EARTH visible to the programmer. While converting sequential code to *any* parallel language requires some effort, the main effort was in conceiving the high-level details of the parallel implementation. Once this was done, the conversion to Threaded-C was relatively straightforward. We believe that once programmers have gained sufficient experience in using Threaded-C, the programming effort is no higher than for MPI.

In conclusion, we have shown that a multithreaded system with local synchronization, overlapping of communication and computation, and low-overhead communication and thread-switching can efficiently parallelize the sparse MVM application.

Acknowledgements

We thank GMD First (Berlin) for research collaboration and for providing us with the MANNA machine used in this study. The authors also acknowledge partial support from DARPA, NSA, and NASA through the HTMT project; NSF (grants CISE-9726388, MIPS-9707125, EIA-9972853, and CCR-9808522); and DARPA through the DIVA project. Agrawal was also supported by NSF CAREER award ACR-9733520.

The authors would like to thank the current and former members of the ACAPS group at McGill University, and the CAPSL group at the University of Delaware, for their insights, ideas, encouragement and help.

References

[1] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[2] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[3] S.T. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NAS Systems Division, NASA Ames Research Center, November 1992.

[4] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[5] Numerical Aerospace Simulation Facility. NAS parallel benchmarks, 1997. http://www.nas.nasa.gov/Software/NPB/.

[6] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 1994. IEEE Computer Society.

[7] Ian Stuart MacKenzie Walker. Towards a custom EARTH synchronization unit. Master's thesis, University of Delaware, Newark, Delaware, July 1999.
[8] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.
[9] Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the EARTH multithreaded architecture. CAPSL Technical Memo 38, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2000. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

On the Predictive Quality of BSP-like Cost Functions for NOWs*

(Extended Abstract)

Mauro Bianco and Geppino Pucci

Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy.
{bianco1,geppo}@dei.unipd.it

Abstract. The Bulk-Synchronous Parallel (BSP) model [16] provides a simple and portable programming discipline that is particularly suitable for coarse-grained parallel systems such as Networks of Workstations (NOWs). In this work we examine the issue of predictability of the BSP cost function for a NOW consisting of SUN workstations connected through a 10Mbps Ethernet network. In particular, we compare the original BSP cost function with a number of newly proposed variants, with the intent of improving predictability by having the cost function encompass those parameters of the hardware/software system which have the largest impact on performance.

1 Introduction

It is widely recognized [15,5] that the quest for a desirable model of parallel programming is made particularly hard by the objective of achieving the following three properties simultaneously: usability, portability and predictability. *Usability* refers to the ease of designing, analyzing, and coding algorithms in the framework provided by the model. *Portability* denotes the ability of compiling and running programs written according to the model over a wide class of target platforms, achieving good performance on each platform. Finally, *Predictability* implies the ability of the model of forecasting performance of a piece of software via an associated cost function. In this paper, we investigate this latter issue for the *Bulk Synchronous Parallel* (BSP) programming model proposed in [16] in the context of low-end parallel systems made of Networks of Workstations (NOWs).

The BSP model provides an abstract machine made of P processors with local memory, connected by a router which implements batch communication via message passing. Computation is divided into phases, named *supersteps*, each terminated by a barrier synchronization. During a superstep, the processors may execute local computation on data held locally at the beginning of the superstep, and/or exchange messages with other processors. The messages sent during a superstep are made available by the router to their destinations only at the beginning of the next superstep.

The running time of a BSP program is obtained by summing the running times of its constituent supersteps. The execution time of a superstep can be expressed as linear cost function which has the following form [14]:

$$T_{ss}(w,h) = w + gh + l \ , \tag{1}$$

* This research was supported, in part, by the Italian CNR, and by MURST under Project *Algorithms for Large Data Sets: Science and Engineering*.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 638–646, 2000.
© Springer-Verlag Berlin Heidelberg 2000

where w is the local computation time and h is the degree of the *relation* realized by the router, that is, the maximum number of bytes sent or received by any processor. Parameters g and l are meant to capture, respectively, the bandwidth and latency characteristics of the underlying architecture.

The simple programming paradigm offered by BSP implies a good level of usability. Also, the inherently machine-independent nature of its communication mechanism, based on batch communication, allows optimized implementations on a large spectrum of parallel architectures, hence fostering efficient portability. However, it has often been noted that the BSP cost function offers only a coarse level of predictability [12,11]. In fact, such observation has motivated further research into defining more descriptive (hence, less usable) models which embody additional aspects of a machine that impact performance (e.g., message injection overhead [6], or clustering [9]). In this paper, we take a different approach. Rather than changing the BSP programming model, we seek to improve its predictability by striving for a tighter coupling between its associated cost function and those features of the hardware/software system under consideration which have the greatest impact on performance. By intervening on the cost function only, we aim at enhancing predictability while preserving usability and portability of the programming model as much as possible.

The programming environment used in this work is based on the message-passing primitives provided by the *BSPlib* library developed by the *Parallel Applications Centre* of Oxford University [10]. *BSPlib* has been installed on a NOW of 10 SUN SPARC-stations available at our department, connected by a 10Mbps Ethernet under the UDP/IP protocol [8]. Under *BSPlib*, interprocessor communication occurs when barrier synchronization is called at the end of each superstep, and is realized through a kind of randomized, time-division multiplexing technique [7]. More specifically, time is divided into *time-slices*, which are in turn divided into as many *time-slots* as the number of sending processors. At each time-slice, the sending processors randomly choose a time-slot for sending their messages over the Ethernet. Randomization helps the system pick a transmission schedule that makes a good usage of the available bandwidth of the communication medium. The time-slot duration depends on the maximum Ethernet frame size supported, and packet fragmentation is done at the library level accordingly. As a consequence, when using *BSPlib* there is a limited payoff in orchestrating communication at the program level so to send one (very) long message rather than many (relatively) short ones, hence we can safely refer only to the total amount of bytes sent from one processor to another.

1.1 Our Contribution

The main purpose of this work is to estimate the relative accuracy and the ease of use of a set of cost functions alternative to the classical BSP function of Equation (1) for the hardware/software system under consideration. Although our quantitative results are system-specific, the proposed methodology is rather general and applicable to a wide range of parallel platforms.

We describe the message routing instance associated to a BSP superstep by means of a *communication pattern*, which can be envisioned as a $P \times P$ array containing, for each processor, the number of bytes that the processor sends to any other processor (including itself). The BSP cost function yields the same prediction for all communication patterns which realize an h-relation. However, h might be too drastic a summary for the characteristics of a communication pattern, hence unsuitable to differentiate among those that have the same value of h but feature very different execution times.

To achieve a more effective (yet simple) categorization, we follow a classic approach in routing theory [13,12] and summarize a communication pattern as an (h_i, h_o, M)-*relation*, where h_i (resp., h_o) is the maximum number of bytes received (resp., sent) by any processor and M is the total number of bytes exchanged by the processors.

The candidate cost functions that we consider are the following linear combinations of the parameters h_i, h_o, M and $h = \max\{h_i, h_o\}$:

1. $\mathcal{F}_h(h) = g \cdot h + l$
2. $\mathcal{F}_{io}(h_i, h_o) = g_i \cdot h_i + g_o \cdot h_o + l$
3. $\mathcal{F}_{ioM}(h_i, h_o, M) = g_i \cdot h_i + g_o \cdot h_o + g_M \cdot M + l$
4. $\mathcal{F}_{hM}(h, M) = g \cdot h + g_M \cdot M + l$
5. $\mathcal{F}_M(M) = g_M \cdot M + l$
6. $\mathcal{F}_{oM}(h_o, M) = g_o \cdot h_o + g_M \cdot M + l$
7. $\mathcal{F}_{iM}(h_i, M) = g_i \cdot h_i + g_M \cdot M + l$
8. $\mathcal{F}_o(h_o) = g_o \cdot h_o + l$
9. $\mathcal{F}_i(h_i) = g_i \cdot h_i + l$

In order to obtain the coefficients for the above cost functions, we execute an extensive set of carefully designed communication patterns, whose objective is to exercise a large number of feasible combinations of the three parameters h_i, h_o and M. The running times collected for such patterns are then used to infer the cost functions through least-square fitting. Finally, the predictive quality of the functions is validated on a suite of additional, synthetic access patterns and on a small set of sorting applications.

2 Fitting the Cost Functions

Consider a P-processor BSP machine, where the i-th processor is denoted by P_i, with $0 \leq i \leq P-1$. Let also $\mathcal{H}_1 = \{h : h = 10000 + 30000 \cdot i, i = 0, \ldots, 3\}$, $\mathcal{H}_2 = \{h : h = 150000 + 75000 \cdot i, i = 0, \ldots, 11\}$ and $\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2$. Finally, let x be an integer parameter.

For each value of $h \in \mathcal{H}$ and $1 \leq x \leq P$, we define the following *synthetic* communication patterns, that will be used for fitting and validating the cost functions:

- (h, x)-`scatter` ($h_o = h$, $h_i = h \cdot x/P$, $M = h \cdot x$). For $0 \leq i \leq x-1$ and $0 \leq j \leq P-1$, P_i sends h/P bytes to P_j.
- (h, x)-`gather` ($h_i = h$, $h_o = h \cdot x/P$, $M = h \cdot x$). For $0 \leq i \leq P-1$ and $0 \leq j \leq x-1$, P_i sends h/P bytes to P_j.
- (h, x)-`square` ($h_i = h$, $h_o = h$, $M = h \cdot x$). For $0 \leq i \leq x-1$ and $P-x \leq j \leq P-1$, P_i sends h/x bytes to P_j, with $j = P-x, \ldots, P-1$.
- `random-`(h, x)`-scatter`. A random communication pattern uniformly generated among all those with $h_o = h$, $h_i = h \cdot x/P$ and $M = h \cdot x$.
- `random-`(h, x)`-gather`. A random communication pattern uniformly generated among all those with $h_i = h$, $h_o = h \cdot x/P$ and $M = h \cdot x$.
- `random-`(h, x)`-square`. A random communication pattern uniformly generated among all those with $h_i = h$, $h_o = h$ and $M = h \cdot x$.

In order to filter out noise, each pattern is executed 20 times and the running time is considered to be the median of the 20 executions. Together, the first three families of patterns (obtained by varying $h \in \mathcal{H}$ and $1 \leq x \leq P$) make up *Suite 1*, which contains *deterministic patterns*, while the last three make up *Suite 2* which is made of *random patterns* sharing the same summary parameters of their deterministic counterparts. Note

	$\mathcal{F}_h$	$\mathcal{F}_{io}$	$\mathcal{F}_{ioM}$	$\mathcal{F}_{hM}$	$\mathcal{F}_M$	$\mathcal{F}_{oM}$	$\mathcal{F}_{iM}$	$\mathcal{F}_o$	$\mathcal{F}_i$
$g \cdot 10^6$	3042			2207					
$g_i \cdot 10^6$		587.2	552.6				1433		2850
$g_o \cdot 10^6$		2932	2898			3066		3386	
$g_M \cdot 10^6$			22.94	334.1	891.5	119.8	530.8		
l	41.57	25.21	26.59	41.47	396.1	67.79	242.6	76.30	280.3

(a) $P = 4$

	$\mathcal{F}_h$	$\mathcal{F}_{io}$	$\mathcal{F}_{ioM}$	$\mathcal{F}_{hM}$	$\mathcal{F}_M$	$\mathcal{F}_{oM}$	$\mathcal{F}_{iM}$	$\mathcal{F}_o$	$\mathcal{F}_i$
$g \cdot 10^6$	6520			4742					
$g_i \cdot 10^6$		644.3	427.1				2336		5699
$g_o \cdot 10^6$		7070	6853			6972		7531	
$g_M \cdot 10^6$			77.61	395.0	994.9	116.4	700.5		
l	104.9	74.51	84.06	104.9	995.1	122.6	702.7	142.8	824.5

(b) $P = 8$

Fig. 1. Cost function coefficients (in $m$secs).

that the Suites exercise a vast spectrum of feasible values of the 3-tuple (h_i, h_o, M), which is crucial to achieve reliable fits. In particular, by varying x, we obtain patterns characterized by a varying amount and distribution of the global communication traffic, but featuring the same value h of maximum outbound/inbound traffic from/to the same processor. Note that scatter-like (resp., gather-like) patterns are likely to incur in higher overhead during message injection (resp., receipt) since $h = h_o \geq h_i$ (resp., $h = h_i \geq h_0$).

We use the two Suites of patterns to fit (over the patterns in Suite 2) and validate (over the deterministic patterns in Suite 1) the BSP-like cost functions defined in the previous sections for two submachines of 4 and 8 processors, respectively. The coefficients of the cost functions obtained for $P = 4$ and $P = 8$ are shown in Fig. 1.

3 Validation Results

The results of the validations of the cost functions on Suite 1 are shown in Fig. 2, where for each submachine and each cost function we report, respectively, the maximum and the average relative errors incurred by approximating the running time of a pattern with the value returned by the function. Note that for $P = 4$, the two-parameter function $\mathcal{F}_{io}$ behaves better, on average, than the three-parameter function $\mathcal{F}_{ioM}$. This counter-intuitive phenomenon can be explained if we consider that the least-square function obtained from the fitting minimizes the $\| \cdot \|_2$-error, while we have chosen to check the quality of our functions against the (more intuitive) metric of relative error between predicted and measured running time. However, when $P = 8$, $\mathcal{F}_{ioM}$ becomes slightly more predictive than $\mathcal{F}_{io}$, which provides evidence that the impact of parameter M becomes more important as P grows.

Also, note that the classical $\mathcal{F}_h$ function is consistently much worse than functions $\mathcal{F}_o$, $\mathcal{F}_{io}$ and $\mathcal{F}_{ioM}$, and that all functions including h_o as a parameter behave decidedly better than those not including it. In retrospect, this behaviour can be explained by the message-scheduling strategy implemented by *BSPlib*[7], where the number of time-

		$\mathcal{F}_h$	$\mathcal{F}_{io}$	$\mathcal{F}_{ioM}$	$\mathcal{F}_{hM}$	$\mathcal{F}_M$	$\mathcal{F}_{oM}$	$\mathcal{F}_{iM}$	$\mathcal{F}_o$	$\mathcal{F}_i$
$P = 4$	Max. Err.(%)	168	17.2	16.4	124	810	99.3	489	120	594
	Ave. Err.(%)	24.7	9.4	9.6	25.7	95.6	18.0	68.1	19.2	78.1
$P = 8$	Max. Err.(%)	425	65.4	62.5	316	559	51.0	379	44.9	461
	Ave. Err.(%)	34.8	7.31	6.80	31.5	86.4	7.54	70.6	7.86	87.7

Fig. 2. Maximum and average validation errors on Suite 1.

slots and time-slices (hence, the duration of the routing) mainly depends on h_o and is independent of h_i.

In addition, we note that when $P = 8$, function $\mathcal{F}_o$ is roughly as predictive as functions $\mathcal{F}_{oM}$, $\mathcal{F}_{io}$ and $\mathcal{F}_{ioM}$, the other parameters embodied by the latter functions having only a second-order effect on improving predictability. Therefore, the simple $\mathcal{F}_o$ function (in fact, even simpler than the classical BSP $\mathcal{F}_h$ function) represents the best compromise between accuracy and simplicity of prediction for a moderately-sized machine. Since the impact of the overall traffic volume (as measured by M) on predictive quality seems to increase with P, it is reasonable to assume that for larger systems, function $\mathcal{F}_{oM}$ would be a better choice.

From our analysis it follows that parameter h_i may be disregarded on our system, since communication time does not seem to depend crucially on the number of messages received by a processor. On the other hand, communication time exhibits a strong linear dependence on parameter h_o. Finally, the synthesis between this two parameters used by the classical BSP function $\mathcal{F}_h$, does not seem to yield good predictions.

In order to fully appreciate the crucial impact of h_o on performance, in Fig. 3 we plot the execution times of some patterns (for varying values of h) in Suite 1, together with all the cost functions under examination, for $P = 8$. Note that when $x = 8$, (h, x)-`scatter`, (h, x)-`gather` and (h, x)-`square` patterns all become *total exchange* patterns, with all processors sending/receiving h/P bytes to/from one another, hence Fig. 3(b) ($(h, 8)$-`gather`) also represents an $(h, 8)$-`scatter` or an $(h, 8)$-`square`. By comparing Figg. 3(a) and 3(b) we note that the running time of an (h, x)-`gather` heavily depends on x, while a comparison of Fig. 3(b) with Figg. 3(c) and 3(d) reveals that there is no such dependency for (h, x)-`scatter` and (h, x)-`square`. Finally, it is very clear from the plots that all functions including h_o as a parameter are much better predictors than the remaining functions, which give rather poor predictions especially for unbalanced patterns (small values of x).

In summary, our experiments imply that one can obtain reliable performance predictions on the hardware/software system under study by adopting a simple variant of the classical BSP cost function, where the contribution of parameter h_o is made explicit. More importantly, we want to point out that *BSPlib* attains such level of predictability while making good use of the hardware, since the peak transmission bandwidth observed during our experiments (8.8Mbps for total exchange patterns) comes close to 90% of the maximum available bandwidth of the communication medium (10Mbps).

4 Predicting the Communication Time of Sorting Algorithms

To test the quality of the above cost functions in real scenarios, we have exercised them on predicting the communication time of *BSPlib* implementations of three classical sorting algorithms, namely, Batcher's *Bitonic Sort* [2]; a simple parallelization of the

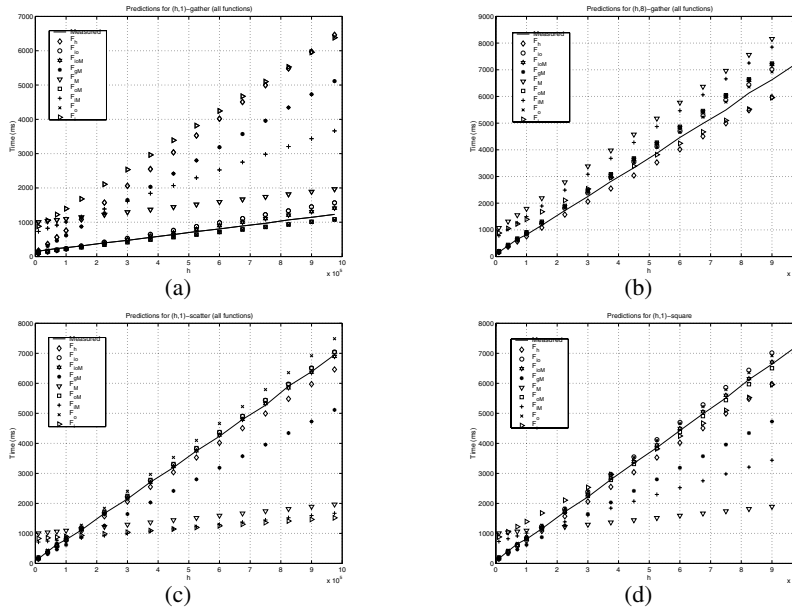


Fig. 3. Running times and predictions for (h, x)-`gather`, (h, x)-`scatter` and (h, x)-`square`, for $x = 1{,}8$. (Fig. 3(b) is also a plot for $(h, 8)$-`scatter` and $(h, 8)$-`square`.)

Radix Sort algorithm for integer sorting; and finally *Sample Sort* with oversampling [4]. We measured the communication times of each constituent superstep by subtracting the time required for local computation from the overall running time of the superstep. (More details on the algorithms will be provided in the full version of this extended abstract.)

Let $\mathcal{N}_1 = \{N : N = 2500 + 7500 \cdot i, i = 0, \ldots, 3\}$, $\mathcal{N}_2 = \{N : N = 37500 + 18750 \cdot i, i = 1, \ldots, 11\}$ and $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$. The sorting algorithms have been executed with random inputs of size $N \cdot P$, for each $N \in \mathcal{N}$, with measured communication times chosen as the median time out of five executions.

The table in Fig. 4 compares the maximum and average prediction errors incurred, for each sorting algorithm, by the BSP function $\mathcal{F}_h$ and the functions that turned out to be better predictors on the synthetic patterns, namely $\mathcal{F}_{io}$, $\mathcal{F}_{ioM}$, $\mathcal{F}_{io}$ and $\mathcal{F}_o$. Also, Fig. 5 plots the measured communication times against the predictions of $\mathcal{F}_h$ (the worst function) and $\mathcal{F}_{ioM}$ (the best function) as a function of N. As before, it is clear that functions including parameter h_o yield much better predictions than $\mathcal{F}_h$, although the difference in quality is not so dramatic as the one observed on the patterns in Suite 1. This relatively better behaviour of the $\mathcal{F}_h$ function is mainly due to the fact that the most expensive communication patterns (at least for radix and sample sort) generated by the sorting applications tend to be total exchanges of N/P^2-length messages, which are those on which $\mathcal{F}_h$ incurs into the least prediction errors, since such patterns have $h = h_i = h_0$, hence coalescing the indegree and the outdegree of the relation into the "summary" parameter h does not imply a large loss of information. Consequently, the improvement of over 50% on the quality of the predictions provided by the more com-

plex $\mathcal{F}_{oM}$ and $\mathcal{F}_{ioM}$ functions can be explained mainly with the presence of parameter M, which captures the impact of the overall traffic volume generated by the pattern.

	Bitonic		**Radix**		**Sample**	
	$P=4$	$P=8$	$P=4$	$P=8$	$P=4$	$P=8$
$\mathcal{F}_h$	43.51	26.52%	17.90	16.43%	22.77	11.10%
$\mathcal{F}_{io}$	35.74	16.15%	13.31	9.93%	17.35	5.01%
$\mathcal{F}_{ioM}$	35.45	13.92%	12.26	6.42%	18.72	4.01%
$\mathcal{F}_{oM}$	36.98	13.62%	11.78	3.17%	42.18	7.37%
$\mathcal{F}_o$	39.69	16.91%	15.24	6.72%	42.91	10.25%

Fig. 4. Average prediction errors for bitonic sort, radix sort and sample sort for $P = 4, 8$ and $n \in \mathcal{N}$

The data collected for bitonic sorting are definitively the most puzzling. Although the communication patterns generated by the algorithm are extremely regular (namely, permutations of N/P-length messages) *all* the cost functions tend to severely underestimate the associated running time. We conjecture that this phenomenon is due to a suboptimal management of this important class of communication patterns by the scheduling algorithm provided by the *BSPlib* library. Note, however, that even in this case, functions embodying the h_o parameter are much better predictors than the BSP function $\mathcal{F}_h$.

5 Future Work

Further investigation is needed to determine to which extent the newly proposed cost functions can be effectively used in practice as an alternative to the classic BSP function to enhance predictability. We believe that the value of separating the contributions of h_i and h_o and adding a parameter of global congestion of the communication medium such as M will prove to be even more substantial for applications characterized by more irregular communication patterns than sorting. In order to substantiate this intuition, we are thinking of exercising our cost functions over a bulk-synchronous version of the *NAS* benchmarks [1].

An orthogonal line of investigation concerns devising cost functions for other network architectures, such as 100Mbps or Gigabit Ethernet, Myrinet or ATM, or comparing the performance/predictability levels achieved by *BSPlib* against those attained by other communication libraries, such as the BSP *PUB* library developed at Paderborn University [3].

Acknowledgments We are grateful to Nancy Amato and Andrea Pietracaprina for helping set the ground for this research. We also wish to thank the EUROPAR referees for their valuable comments and suggestions.

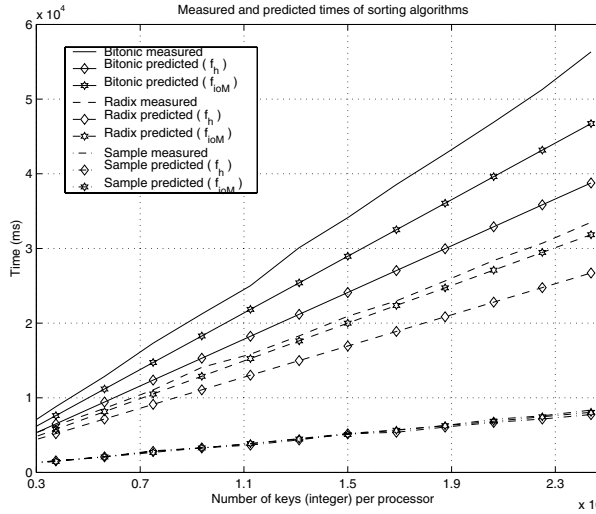


Fig. 5. Measured versus predicted (functions $\mathcal{F}_h$ and $\mathcal{F}_{ioM}$) communication times for Bitonic, Radix and Sample sort.

References

1. D. Bailey, E. Barszcz, J.T. Barton, *et al.* The NAS parallel benchmarks. *Int. J. Supercomputer Appl.*, 5(3):63–73, 1991.
2. K.E. Batcher. Sorting networks and their applications. In *Proc. of the AFIPS Spring Joint Computer Conf.*, pages 307–314, 1968.
3. O. Bonorden, B. Juurlink, I. von Otte, and I Rieping. The Paderborn university BSP (PUB) Library – Design, Implementation and Performance In *Proc. of the 2nd merged IPPS/SPDP Symp.*, pages 99-104, San Juan, Puerto Rico, April 1999.
4. G. Blelloch, C.E. Leiserson, B.M. Maggs, *et al.* A comparison of sorting algorithms for the connection machine CM-2. In *Proc. of the 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head SC, USA, 1991.
5. G. Bilardi, A. Pietracaprina, and G. Pucci. A quantitative measure of portability with application to bandwidth-latency models for parallel computing. In *Proc. EURO-PAR'99 – Parallel Processing*, pages 543–551, Toulouse, F, Aug./Sep 1999.
6. D.E. Culler, R. Karp, D. Patterson, *et al.* LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.
7. S.R. Donaldson, J.M.D. Hill, and D.B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP. In *Proc. EURO-PAR'98 – Parallel Processing*, pages 970–980, Southampton, UK, September 1998.
8. S.R. Donaldson, J.M.D. Hill, and D.B. Skillicorn. BSP clusters: high performance, reliable and very low cost. Technical report PRG-TR-5-98, Oxford University Computing Laboratory, Oxford, UK, 1998.
9. P. De la Torre and C.P. Kruskal. Submachine locality in the bulk synchronous setting. In In *Proc. EURO-PAR'96 – Parallel Processing*, pages 352–358, Lyon, F, August 1996.
10. M. Goudreau, J.M.D. Hill, W. McColl, S. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library. BSP Worldwide `http://www.bsp-worldwide.org/`, April 1996.
11. M. Goudreau, K. Lang, S. Rao, *et al.* Portable and efficient parallel computing using the BSP model. *IEEE Trans. on Computers*, C-48(7):670–689, July 1999.

12. B.H.H. Juurlink and H.A.G. Wijshoff. A quantitative comparison of paralle computation models. *ACM Trans. on Computer Systems*, 16(3):271–318, 1998.
13. F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.
14. W.F. McColl. BSP programming. *DIMACS Series in Discrete Mathematics*, pages 21–35. American Mathematical Society, 1994.
15. B. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. of the 28th Hawaii Int. Conf. on System Sciences (HICSS)*, volume 2, pages 61–70, January 1995.
16. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs*

Siegfried Benkner[1] and Thomas Brandes[2]

[1] Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
sigi@ieee.org
[2] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
brandes@gmd.de

Abstract. The OpenMP Application Program Interface supports parallel programming on scalable symmetric multiprocessor machines (SMP) with a shared memory by providing the user with simple work-sharing directives for C/C++ and Fortran so that the compiler can generate parallel programs based on thread parallelism. However, the lack of language features for exploiting data locality often results in poor performance since the non-uniform memory access times on scalable SMP machines cannot be neglected. HPF, the de-facto standard for data parallel programming, offers a rich set of data distribution directives in order to exploit data locality, but has mainly been targeted towards distributed memory machines. In this paper we describe an optimized execution model for HPF programs on SMP machines that avails itself with the mechanisms provided by OpenMP for work sharing and thread parallelism while exploiting data locality based on user-specified distribution directives. This execution model has been implemented in the ADAPTOR HPF compilation system and experimental results verify the efficiency of the chosen approach.

1 Introduction

There is now an emerging class of multiprocessor architectures with scalable hardware support for cache coherence. These are generally referred to as (scalable) Shared Memory Multiprocessor (SMP) architectures. Most of these machines are built via physically distributed memory (ccNUMA) resulting in non-uniform memory access times and therefore the exploitation of data locality is a crucial issue for many applications.

The OpenMP Application Programming Interface [15] is intended as a portable shared memory programming model to be used on SMP architectures. It defines a set of program directives and a library for runtime support that augment

* The work described in this paper was supported by NEC Europe Ltd. as part of the ADVICE project in cooperation with the NEC C&C Research Laboratories.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 647–657, 2000.
© Springer-Verlag Berlin Heidelberg 2000

standard C/C++ and Fortran 77/90. OpenMP is based on thread parallelism allowing users to exploit shared memory parallelism at a reasonable coarse level. But OpenMP does not provide any directives for controlling the locality of data. As a consequence, the user is responsible for achieving high data locality by enforcing an appropriate work and data distribution which results in a significantly higher programming effort.

High Performance Fortran (HPF) [9] is a well established language extension of Fortran supporting the data parallel programming model. While Fortran array statements and the `FORALL` statement are already natural ways of specifying data parallel computations, HPF provides additional directives to assert independent computations and to advise the compiler how to assign array elements to processor memories in order to reduce data movements on machines with Non-Uniform-Memory-Access (NUMA). The HPF mapping directives define a mapping of data objects (arrays) to abstract processors. Data objects that have been mapped to a certain abstract processor are said to be *owned* by that processor. *Ownership* of data is the central concept for the execution of HPF programs. Based on the ownership of data (owner-computes rule), the distribution of computations to the abstract processors and the necessary communication and synchronization between processors is derived automatically. Up to now, the compilation and execution of HPF programs is mainly considered for distributed memory (DM) machines based on a DM execution model, illustrated in Figure 1(b). In this model, the parallel program generated by the compiler is executed by a set of (abstract) processors where each processor executes the same program in its local address space operating only on its own data. Any two processors communicate by exchanging messages. In accordance with the SPMD (single-program-multiple-data) paradigm, a HPF compiler has to ensure that all processors executing the target program follow the same control flow in a loosely synchronous style. Scalar data and data without mapping directives are allocated on each processor, i.e. replicated.

In this paper, we present and discuss a highly-efficient execution model for SMP architectures. It is illustrated in Figure 1(c). In contrary to the DM execution model that can also be emulated on SMP machines, it takes advantage of the global address space provided on these machines in order to generate more efficient code. While the work distribution implied by the mapping directives remains the same as in the DM model, the model uses thread parallelism instead of process parallelism and keeps the global layout of the mapped data in the global address space to reduce the overhead for non-local data accesses. Scalar data and data without mapping directives have only one incarnation in the shared memory in order to reduce memory overheads.

The SM execution model described in this paper has been implemented within the public domain HPF compilation system ADAPTOR [1]. This compilation system already supported the DM execution model for a long time. It has been redesigned to support both execution models in such a way that most of the compiler modules can be exploited for both models. The user can select the execution model by specifying a flag [5]. For the SM execution model, ADAPTOR

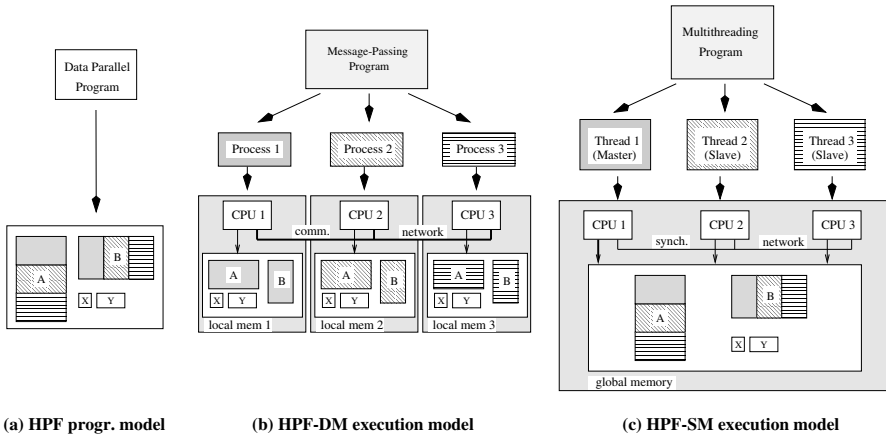


Fig. 1. HPF execution model for distributed and shared memory machines.

generates Fortran code with embedded shared memory parallelization directives exploiting thread parallelism (e.g. OpenMP, NEC/SX or SGI Fortran directives). Additional runtime support is available to support data locality.

The experimental results for some applications show that on SMP architectures the SM execution model is more efficient than the DM execution model. This is especially true for unstructured applications with indirect addressing of distributed arrays where the DM model requires complex run time support for the calculation of communication patterns and where the SM model allows direct access to the shared data.

Related Work

Many authors have addressed certain issues involved in the exploitation of HPF-like data parallelism via an SMPD execution model on shared memory systems, especially for optimizing synchronization (e.g. [11, 8]).

On the Origin2000, the SGI data placement directives [14] form a vendor specific extension of OpenMP. Some of the directives have similar functionality as the HPF directives, e.g. the "affinity scheduling" of parallel loops is the counterpart to the `ON` clause of HPF.

Chapman, Mehrotra and Zima [6] propose a set of extensions to OpenMP to provide support for locality control of data, similar to HPF mapping directives, but they do not provide a detailed execution model or implementation scheme.

Portland Group, Inc. proposes a new high-level programming model [10] that extends the OpenMP API with additional data mapping directives, library routines and environment variables. This model offers more capabilities to direct locality of data and is also applicable for distributed memory systems and SMP clusters. In contrary to this model, we rely on a uniform data parallel programming model suited for all kinds of architectures.

2 Thread Parallelism vs. Process Parallelism

For the DM execution model, the HPF compiler generates a program to be executed by a set of processes where each process executes the same program in its local address space. This corresponds to the Single-Program-Multiple-Data (SPMD) paradigm that is also used in the message passing programming model. All processes follow the same control flow in a loosely synchronous style. Usually there is a one-to-one mapping of abstract processors (declared at the HPF level by means of processors directives) to processes and each process is executed on an individual processor of the parallel target architecture.

Light-weighted processes (threads) are based on the idea of separating the characteristics of program execution (program counter, stack) from system resources (memory, table of open files). One process can split up in many threads that have their own stack and program counter, but share all the other system resources. Operations on threads (creation, context switching) are executed at least one order of magnitude faster than the corresponding operations on processes. Threads are the first choice when using the master/slave paradigm that is also followed within the OpenMP programming model.

For our SM execution model we chose to generate programs based on thread parallelism. The main reason for this decision is the fact that the global address space is available by default and does not have to be allocated via special system calls. Nevertheless, we follow the SPMD paradigm as far as possible to avoid the overhead caused by the creation/termination of threads. Attention has been paid to efficient memory allocation during parallel execution as it must be thread-safe.

3 Data Mapping and Data Layout

For the SM execution model, we changed the default mapping strategy for scalar data and non-mapped data. While in the DM model every processor is an owner and gets an own copy of this data, we have now only one incarnation in the global address space that will be owned by a dedicated processor (master thread). This strategy for the SM model reduces the memory overhead but might cause some additional synchronizations. In any case, the user can still replicate explicitly the data to have the same behavior as in the DM model.

For the mapped data, the HPF mapping directives define a mapping of data to the abstract processors. This mapping defines the ownership of data. In the DM execution model, mapped data objects are allocated in a partitioned manner, such that each processor only allocates those parts of a data object that are *owned* by it. The part of a distributed array owned by a processor is referred to as its *local section*. Since the size of the local section of a distributed array on a particular processor usually cannot be determined at compile time, a dynamic allocation strategy has to be adopted. Since each processor allocates only a subsection of a distributed array, global addresses of distributed arrays have to be translated into local addresses. In particular for cyclic and indirect distributions, these additional local address calculations may add significant overhead.

On shared memory machines, the availability of a global address space obviates the need to allocate mapped HPF data objects in a partitioned way. As a consequence, the original global Fortran layout can be left unchanged. The global address space offers several advantages regarding the execution of data parallel programs: translation of global addresses to local addresses is superfluous and remapping of data changes only the ownership of data but does not imply any reallocation of data or data movements in memory.

In certain situations, however, the global data layout might decrease the cache performance due to false sharing. In such cases, a reorganization of distributed arrays such that all array elements belonging to the same processor are stored contiguously might be more efficient. We provide a new compiler specific directive to specify an attribute `LOCAL_LAYOUT` for mapped arrays that implies such a data layout.

4 Work Distribution

Parallel execution in the HPF execution model is achieved by distributing the computations to the processors. This task, referred to as work distribution, is usually based on the owner-computes rule where an assignment to an element of a distributed array is executed by the processor that owns this element. Thus, work distribution is derived automatically from the data distribution. The owner computes rule is not always the best strategy for work distribution. Therefore alternative strategies may be adopted by an HPF compiler. Moreover, HPF-2 provides the `ON` clause that allows the user to explicitly control work distributions.

For the SM execution model, we propose the same work distribution strategy as in the DM execution model, i.e. the work sharing of parallel loops is derived automatically from the data distribution of arrays accessed within the loop and implemented by means of appropriate OpenMP work sharing constructs. There is only one difference due to the changed default allocation strategy for scalars and non-mapped arrays. Assignments to such data objects are no longer executed by all processors (replicated) but only by one dedicated processor (master thread).

5 Communication and Synchronization

By analyzing the data distribution and work distribution, non-local data accesses can be determined. These non-local data accesses imply data movements between the involved processors.

In the DM execution model, non-local accesses are implemented by means of message-passing. As a consequence, each processor has to determine both the data to be sent to other processors and the data to be received from other processors. Since all cross-processor dependencies are satisfied in this manner, no explicit synchronization is necessary. In the context of parallel loops, communication is extracted from the loop body combining single-element messages into larger messages whenever possible in order to minimize latency (message

vectorization). In case of indirect addressing, an inspector/executor strategy [13] should be exploited to derive a communication schedule. Temporary arrays and buffers become necessary to keep non-local data. The introduction of shadow edges reduces this memory overhead and simplifies the generated code.

In the SM execution model, a thread can access non-local data directly via the global address space. Only appropriate synchronization becomes necessary to ensure that the correct version of the shared data is accessed. This synchronization can be realized either by point-to-point synchronization between the accessing and the owning processor or by global synchronization via barriers. Synchronization is extracted from parallel loops utilizing similar techniques as applied for communication optimization in the DM model. Our straightforward approach inserts a barrier before and after every independent computation with non-local accesses (e.g. see Figure 2) avoiding buffers for non-local data and expensive computation of communication schedules in case of indirect addressing. A lot of techniques are known to replace barrier synchronization with cheaper producer-consumer synchronization (e.g. [11]) and for reducing synchronization costs (e.g. [8]). These techniques might be exploited in an advanced compilation system.

The compiler will not generate any synchronization for the SM execution model if there is not any interprocessor communication for the DM model. In the same way as the `RESIDENT` directive of HPF allows avoiding redundant interprocessor communication or related overhead for possible communication it is utilized in the SM model to avoid unnecessary synchronization.

```
      integer, dimension (N) :: K
      real, dimension (N) :: A, B
!hpf$ distribute (block)  :: A, B


      ...
      B = A(K)
      B = B - A
      A = A + B
      ...
```

(a) original HPF code.

```
!$omp parallel, private (I,LB,UB)
      LB = ...; UB = ...! local range
      ...
!$omp barrier
      do I = LB, UB
         B(I) = A(K(I))
      end do
!$omp barrier
      do I = LB, UB
         B(I) = B(I) - A(I)
         A(I) = A(I) + B(I)
      end do
      ...
!$omp end parallel
```

(b) generated OpenMP code.

Fig. 2. Synchronization of non-local accesses.

6 Private and Reduction Variables

Private variables specified by the `NEW` clause of HPF become private data of the threads in the SM execution model. Thus every thread will get its own local incarnation of the variable. In the DM execution model, every process has its own incarnation by default and the `NEW` clause only guarantees that the different incarnations do not have to be consistent and can therefore have different values.

Reduction variables specified by the `REDUCTION` clause of HPF are treated like private reductions when they are not mapped. Every processor gets an own copy and the values are accumulated at the end of the independent computations. This is the same strategy for both models, only the accumulation is handled differently, either by message passing via a global reduction or by accumulating the result to the shared incarnation within a critical section. A tree-like accumulation in the SM model is supported by the HPF run time system and might be more efficient for a larger number of processors.

Reduction variables that are mapped arrays are treated differently. In the DM execution model, the processors allocate non-local copies for those items of the reduction array that they update. The non-local copies will be accumulated after the independent computation. In the SM execution model, mapped reduction arrays will have only one shared incarnation where the reductions on it are synchronized by locks to ensure that a specific memory location is updated atomically and not accessed simultaneously by multiple writing threads.

In order to minimize synchronization overheads for the reduction array, the concept of *exclusive ownership* has been introduced [3]. An element of the reduction array is exclusively owned by an abstract processor (thread) if it is owned by that processor and not updated by any other processor. Synchronization is only necessary for those elements of the reduction array that are not exclusively owned while exclusively owned elements can be handled like private data requiring no synchronization. The concept of exclusive ownership is especially important for unstructured reductions that are employed in many scientific applications to implement computations on unstructured meshes or sparse matrices.

7 Experiments and Results

ADI-Kernel with Redistributions

The three-dimensional ADI kernel is a data parallel HPF program which utilizes five three-dimensional arrays of size $64 \times 64 \times 64$. The HPF code uses dynamic arrays and redistributes the five arrays from `(*,*,BLOCK)` to `(*,BLOCK,*)` and vice versa for every of the 10 iterations. Only the redistribution requires communication in the DM execution model. Table 1 shows the performance results of the ADI kernel on the NEC SX-4. It compares the data parallel HPF code compiled by ADAPTOR for the different execution models. The DM execution model shows some small speed-ups, but the gain of parallelism in the computations is mainly lost by the time needed for the redistributions. The SM model utilizes

a shared global layout of the arrays, therefore the redistributions only require a synchronization, but no communication and no data movement in memory.

	NP = 1	NP = 2	NP = 4	NP = 8
HPF-DM	1.12 s	1.07 s	0.85 s	0.80 s
HPF-SM	1.14 s	0.57 s	0.30 s	0.16 s

Table 1. Execution times of ADI Kernel on NEC SX-4.

Relaxation on an Unstructured Grid

The next example is a relaxation kernel that uses indirect addressing on an unstructured grid. An integer array specifies for every grid point its four neighbors. The grid data is block distributed, the numbering of the grid points exploits a high data locality. Table 2 gives the execution times for a fixed problem size and different number of processors. Both versions scale, but the SM version is nearly two times faster than the DM version. The DM version requires complex compile time and run time support to compute the communication schedules implied by the indirect addressing. Even when the overhead is amortized over the iterations by reusing the communication schedule, it remains non-negligible. This is especially true on the NEC SX-4 where indirect access to shared arrays can be vectorized but not the computation of the communication schedule.

	NP = 1	NP = 2	NP = 4	NP = 8
HPF-DM	4.327 s	1.972 s	1.021 s	0.545 s
HPF-SM	1.940 s	0.983 s	0.500 s	0.264 s

Table 2. Unstructured relaxation on NEC SX-4 (1M grid points, 73 iterations).

Crash Simulation Kernel

For the evaluation of a scientific computation with unstructured reductions, we used a kernel from an industrial crash simulation code [7]. The kernel is based on a time-marching scheme to perform stress-strain calculations on a finite-element mesh consisting of 4-node shell elements. In each time-step elemental forces are calculated for every element of the mesh and added back to the forces stored at nodes by means of unstructured reduction operations. Besides the computation of elemental forces, the unstructured reduction operations to obtain the nodal forces represent the most important contribution to the overall computational costs. Table 3 shows the elapsed times measured on an SGI Origin 2000 (MIPSpro Fortran compiler, version 7.30) for different variants of a crash kernel performing 100 iterations on a mesh consisting of 25600 elements and 25760 nodes. In the table the entry HPF-DM refers to the HPF version compiled by ADAPTOR for the DM model using the inspector/executor strategy, HPF-SM to the HPF version

NP	1	2	4	8	16	32
`HPF-DM`	6.39	3.58	1.76	0.99	0.61	0.40
`HPF-SM`	5.10	2.79	1.47	0.74	0.55	0.34
OpenMP	11.39	6.51	3.48	2.06	1.41	1.27

Table 3. Execution times (secs) for crash simulation kernel on the SGI Origin 2000.

compiled for the SM execution model using the exclusive-ownership technique, and *OpenMP* to an OpenMP version which synchronizes all updates on the reduction array.

The irregular mesh used in this evaluation exhibits a high locality. As a consequence, both the DM version and the SM version that exploit data locality show very satisfying results. The version using atomic updates for all assignments to the node array scales but exhibits an overhead of about a factor of two.

8 Summary and Conclusion

In this paper we presented an execution model for data parallel programs that takes advantage of threads and the global address space provided on SMPs. Compared to the DM execution model being emulated on SMPs, it avoids the memory overhead implied by replication of data and by non-local copies of data and it yields better performance for a range of applications.

In comparison to the OpenMP programming model, data locality specified in the data parallel HPF program can be exploited directly to achieve better scalability on larger SMPs. Though OpenMP also allows SPMD style parallelism in which the programmer explicitly partitions work and synchronizes the processors correctly to achieve comparable performance with optimized message passing versions, this style results in higher programming effort. If the scheduling clauses of OpenMP are not conform with the chosen data distribution, the distribution of loop iterations among processors must be done by hand. The absence of necessary synchronization results in incorrect programs that are difficult to debug. Furthermore, data parallelism within array statements and `FORALL` statements is exploited within HPF, but only considered for the next version 2.0 of OpenMP. Efficient OpenMP reduction is currently only defined for scalars, reductions on an array must be carried out "by hand" using other synchronization mechanisms (e.g. a critical section for private reductions or the `atomic` directive for shared reductions).

OpenMP still offers some features that make this programming model more attrative than HPF for a certain range of applications: OpenMP allows to create tasks dynamically and provides features to deal with fluctuating workload, and in OpenMP, tasks might interact with each other in non-trivial ways. But recent developments show that these issues can also be addressed in HPF [2, 4]. Nevertheless, dynamic scheduling strategies show non-negligible overhead, e.g. see [12], and non-trivial task interaction reduces scalability.

In contrast to other approaches where the HPF and OpenMP programming model are combined, we rely on a uniform programming model that is appropriate for all architectures. By a hierarchical mapping of data, this programming model can be also exploited on clusters of SMPs. The first hierarchy defines the mapping of data to the nodes, the second one defines the ownership for the processors within the node. The implementation of such an execution model coupling the DM and SM execution model hierarchically is currently under work.

References

[1] ADAPTOR. High Performance Fortran Compilation System. WWW documentation, Institute for Algorithms and Scientific Computing (SCAI, GMD), 1999. `http://www.gmd.de/SCAI/lab/adaptor`.

[2] G. Antoniu, L. Bougé, R. Namyst, and C. Perez. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. In *The 1999 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 4*, pages 1756–1762, Las Vegas, NV, June 1999.

[3] S. Benkner and T. Brandes. Efficient Parallelization of Unstructured Reductions on Shared Memory Parallel Architectures. In *Parallel and Distributed Processing, Proceedings of 15 IPDPS 2000 Workshops, Cancun, Mexico*, Lecture Notes in Computer Science (1800), pages 435–442. Springer Verlag, May 2000.

[4] T. Brandes. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. In Amestoy, P. and Berger, P. and Dayde, M. and Duff, I. and Giraud, L. and Frayssé, V. and Ruiz, D. (Eds.), editor, *Euro-Par'99 Parallel Processing, Toulouse*, pages 833–844. Lecture Notes in Computer Science (1685), Springer-Verlag Berlin Heidelberg, Sept. 1999.

[5] T. Brandes and R. Höver-Klier. ADAPTOR User's Guide (Version 7.0). Technical documentation, GMD, Dec. 1999. Available via anonymous ftp from `ftp.gmd.de` as `gmd/adaptor/docs/uguide.ps`.

[6] B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with Features for Locality Control. Technical report TR99-02, Inst. for Software Technology and Parallel Systems, U. Vienna, Feb. 1999. `www.par.univie.ac.at`.

[7] J. Clinckemaillie, B. Elsner, and G. L. et al. Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.

[8] M. Gupta and E. Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 322–332. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, Jan. 1997.

[10] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP – A Programming Model for SMP Clusters. In *Eighth International Workshop on Compilers for Parallel Computers*, pages 229–238, Aussois, France, Jan. 2000.

[11] M. O'Boyle and F. Bodin. Compiler reduction of synchronisation in shared virtual memory systems. In *9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 318–327. ACM Press, July 1995.

[12] M. Resch, I. Loebich, and B. Sander. A comparison of OpenMP and MPI for the parallel CFD test case. In *Workshop on OpenMP (EWOMP'99) at Lund/Sweden, September 30 - October 1 1999*, Oct. 1999.

[13] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[14] Silicon Graphics, Inc. MIPSpro (TM) Power Fortran 77 Programmer's Guide. Document 007-2361-007, SGI, 1999.

[15] The OpenMP Forum. OpenMP Fortran Application Program Interface. Proposal Ver 1.0, SGI, Oct. 1997. `http://www.openmp.org`.

The Skel-BSP Global Optimizer: Enhancing Performance Portability in Parallel Programming

Andrea Zavanella

Dipartimento di Informatica
Università di Pisa - Italy
zavanell@di.unipi.it
http://www.di.unipi.it/∼zavanell

Abstract. The paper describes the Skel-BSP Global Optimizer (GO), a compile-time technique tuning the structure of skeletal programs to the characteristics of the target architecture. The GO uses a set of optimization rules predicting the costs of each skeleton. The optimization rules refer to a set of implementation templates developed on top of the EdD-BSP (a variant of the BSP model). The paper describes the Program Annotated Tree representation and the set of transformation rules utilized by the GO to modify the starting program. The optimization phases: balancing, scaling and augmenting are presented and explained running the GO on a cluster of PCs for an image analysis toy-program.
key words: skeletons, BSP, optimization, performance portability.[1]

1 Introduction

The Skel-BSP [14] approach has been proposed to conjugate Skeletons [3, 5] and BSP [11] to obtain high level programming and performance portability. The paper presents the Global Optimizer (GO) a compile-time technique tuning Skel-BSP programs to the target architecture. GO uses a set of transformation rules preserving the program semantics and chooses the distribution of processors among the program components. The paper describes the "global" approach utilized by Skel-BSP on top of the "local" optimizations embedded in each implementation template [15, 12, 13]. These rules are based on a BSP-like computational model: the EdD-BSP (see Section 2.2). The paper shows how these two strategies work together to optimize the EdD-BSP intermediate code on a given parallel platform. The GO is presented by describing its main procedures: `augmenting, balancing` and `scaling`. An example of the GO behavior is provided compiling an image analysis toy-program on a cluster of PCs (Backus).

[1] This work has been supported by the Italian M.U.R.S.T. within the Mosaico framework

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 658–667, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 The Skel-BSP Methodology

2.1 The Skel-BSP Compiler

Skel-BSP forces the programmer to concentrate on exposing a "parallelization strategy" more than a parallel algorithm. Therefore the programmer expertise is exploited in the direction of writing a composition of already defined parallel patterns (`Pipe,Farm,Map,Reduce`) according to the P3L programming style [4, 10]. The Skel-BSP compiler derives an optimized implementation using three additional sources:

- a set of BSP-lib [8] implementation templates ,
- a set of performance equations [14],
- the EdD-BSP parameters of the target architecture,

The "local" optimizations are stored in a set of reusable components (templates) with associated two optimization rules expressed by the following equations:

- $T_{opt}(param, M)$: the optimal service time on a given EdD-BSP computer M (see Section 2.2);
- $N_{opt}(param, M)$: the minimal number of workers obtaining the optimal service time on M;

The tuple of application dependent parameters ($param$) is computed using a sequential profiling while the EdD-BSP parameters one (M) is provided by the parallel profiling phase (see Fig. 1).

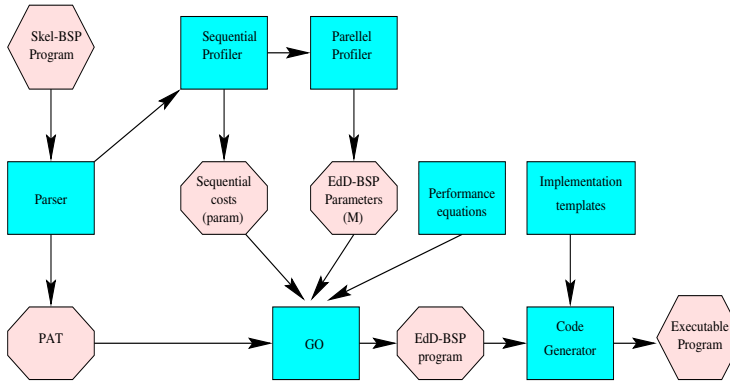


Fig. 1. The structure of the Skel-BSP compiler

2.2 The Cost Model

The EdD(Edinburgh-Decomposable)-BSP model , has been introduced as a variant of the BSP [11] to predict skeletal programs performance. A BSP computer is a set of p couples processor-memory interconnected in order to be able to communicate point-to-point and to perform a global synchronization. A BSP computation is organized as a sequence of synchronous supersteps including (a) a local computation phase, (b) a global communication phase and (c) a barrier synchronization. The cost of each superstep is given by: $T_{sstep} = W + hg + L$ where W is the maximum amount of work performed in the local computation phase, and h is the maximum number of messages sent or received during the communication phase. The parameters g and L are the "standard" BSP parameters defined as the costs to send a single message (g) and to perform a barrier synchronization (L). The EdD-BSP variant introduces two extensions of the BSP model: a couple of parameters g_∞ and $N_{1/2}$ in place of g modeling the communication bandwidth as a function of the message size (see [9], and the decomposability (see the work of Kruskal et al.[6]). An EdD-BSP computer is then a tuple parallel M including four parameters:

$$M = (l, g_\infty, N_{1/2}, p)$$

A relevant innovation introduced by the second extension is the possibility of `partitioning` a BSP computer in submachine. Each submachine acts as an autonomous BSP computer (i.e. it synchronizes independently). The model admits two kinds of supersteps: the computational supersteps, and join/partition supersteps which costs are stated in the following equations:

$$T_{sstep} = \begin{cases} W + hg_\infty(\frac{N_{1/2}}{h} + 1) + L & \text{computational step} \\ L & \text{join-partition} \end{cases}$$

Assuning that at a given time the p processors are partitioned in $q < p$ submachines, the cost to perform a superstep is expressed as the maximum cost for each submachine to reach the next join operation. This means that T_{sstep} has to be computed recursively as the maximum time to execute the EdD-BSP program running on the i machine. Assuming that no other partition is executed we would obtain:

$$T_i = \sum_{j=1}^{nstep(i)} T_{sstep}(i, j)$$

Where $nstep(i)$ is the number of supersteps performed by the submachine i and $T_{sstep}(i, j)$ is the "classic" BSP cost for the j-th superstep of the i-th submachine. This extension enables EdD-BSP to predict the execution costs of skeletal programs whose components require different number of synchronizations. A practical implementation of a decomposable BSP programming has been realized in the Paderborn University BSP library (PUB) [2]. The need for the EdD-BSP model and the results of predicting the cost of skeleton programs using such a model are included in [15, 12, 13].

3 The Program Annotated Tree (PAT)

The Program Annotated Tree is an extension of the syntax-tree of a Skel-BSP program, where each node includes three fields: $(Skel(param), N_w, T_{serv})$. The $Skel(param)$ field contains a skeleton identifier $(Skel)$ and the list $(param)$ of performance parameters (i.e. in Fig. 2 the sizes of input output structures d_0, d_1, d_2). The field N_w contains the number of processors used by the module and T_{serv} is the service time of the subprogram rooted at the node. An example of a node of the PAT is shown in Fig. 2. The initial values stored in the PAT $param$ fields are computed by the sequential profiling phases while the other fields are filled by the GO during the `init` phase.

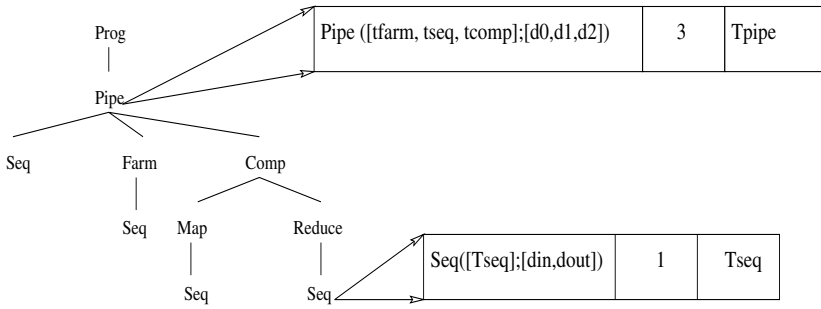


Fig. 2. The PAT of a Skel-BSP program

4 The Global Optimizer

4.1 The Transformation Rules

The GO transforms valid Skel-BSP programs (see the grammar in [14]) using the transformation rules in Tab. 1 to adapt the program to the characteristics of the target machine. The rules 3 and 4 refer to the `Comp` constructor which models the sequential composition of Data Parallel modules. The rule 5 uses the `concat` operator which makes a monolitique sequential constructor from a sequence $X_1, \ldots X_k$ of sequential modules. Fig. 3 shows an example of two valid transformations preserving the semantics of the starting program. The two programs result from two different sequences of program transformations: `(b)` is obtained from `(a)` using 1-3-6, while `(c)` is obtained using 2-7-5.

4.2 Initializing the PAT

The overall structure of GO is shown in Fig. 4. The `init` procedure computes the values of N_w and $Tserv$ for the leaves of the PAT then propagates the results up

Num.	Rule	Name
1	$Seq \longrightarrow Farm(Seq)$	Farm insertion
2	$Farm(Seq) \longrightarrow Seq$	Farm elimination
3	$Comp(X_1, \ldots X_k) \longrightarrow Pipe(X_1, \ldots X_k)$	Pipe insertion
4	$Pipe(X_1, \ldots X_k) \longrightarrow Comp(X_1, \ldots X_k)$	Pipe elimination
5	$Pipe(X_1, \ldots X_k) \longrightarrow concat(X_1, \ldots X_k)$	Pipe collapse
6	$Pipe(X_1, Pipe(Y_1, \ldots Y_h), \ldots X_k) \longrightarrow Pipe(X_1, Y_1, \ldots Y_h, \ldots X_k)$	Pipe fusion
7	$Pipe(X_1, Y_1, \ldots Y_h, \ldots X_k) \longrightarrow Pipe(X_1, Pipe(Y_1, \ldots Y_h), \ldots X_k)$	Pipe distribution

Table 1. The GO transformation rules

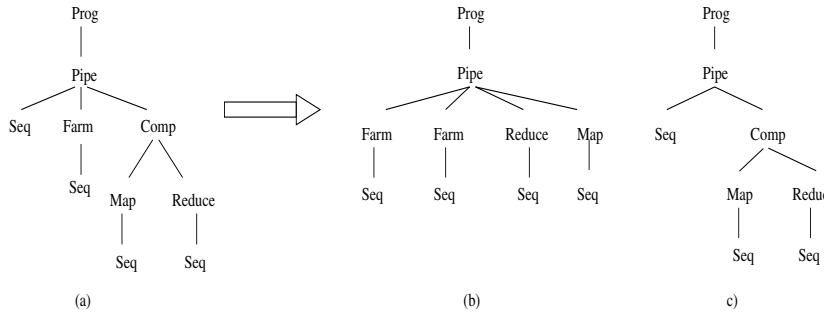


Fig. 3. Two transformations of a valid program

to the root. In this phase the optimization rules for each skeleton are computed on the target M_∞ which allows to saturate each parallel components with the maximum useful parallellism: $M_\infty = (l, g_\infty, N_{1/2}, \infty)$. The PAT for M_∞ is called *fully parallel version* of the program. The pseudo-code of the `init` procedure is given in Fig. 5. The first loop computes the values of T_{serv} and N_w for: Farm, Map, Reduce and Scan. The functions `Topt(Skel,`M_∞`)` and `Nopt(Skel,`M_∞`)` return the optimal values according to the optimization rules defined in [12, 15]. The second loop propagates the values of T_{serv} and N_w to the higher layers of the tree. We have three optimization cases:

1. $N_w > p$: GO reduces the number of processors minimizing the loss of performance;
2. $N_w \leq p$: GO improves the program performance by adding processors;
3. $N_w = p$: GO terminates;

4.3 Reducing Resources

The goal of this phase is to reduce the number of processors to match the number of available processors while minimizing the loss of performance. The reduction takes place in two subphases:

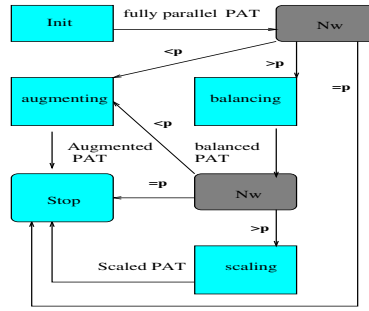


Fig. 4. The GO structure

```
For all Node in PAT such that:
       ((Node.Skel=Farm or
         Node.Skel=Map or
         Node.Skel=Reduce or
         Node.Skel=Scan or
     Node.Skel=Comp) and
         marked(Node)=false)
       Node.Nw=Nopt(Node.skel,M)
       Node.Tserv=Topt(Node.skel,M)
       mark(Node))
endfor (phase 1)
```

```
For all Node in PAT such that:
  ((Node.Skel=Pipe or
        Node.Skel=Comp) and
        marked(children(Node)) and
        marked(Node)=false)
  Update(Node.Skel)
      Node.Nw=Nopt(Node.skel,M)
      Node.Tserv=Topt(Node.skel,M)
      mark(Node)}
endfor
(phase 2)
```

Fig. 5. The two phases of the GO init algorithm

1. *balancing*: in this phase the number of processors employed by each pipeline stage with service time $T_{serv} < T_{slow}$ (where T_{slow} is the service time of the slowest stage) is "reduced".
2. *scaling*: when the balancing phase is completed, if N_w is still larger than p, the program must be scaled.

The `balance` procedure computes the minimum number of processors mp such that: $Tserv \leq T_{slow}$. In the case: $mp = 1$ the suitable elimination rule is applied and the PAT is transformed by setting: $Node.skel = Seq$. At the end of the balancing phase, three optimization cases may occur:

1. $N_w = p$ GO terminates;
2. $N_w < p$ GO performs the augmenting phase;
3. $N_w > p$ GO performs the scaling phase;

The difficulty of scaling the program arises from the fact that many transformations may reduce the number of processors and GO must select which one leads to the optimum. Other works have demonstrated [1] that using a gradient method the optimizer may stop in a local optimum while the global optimum

could have been reached by accepting not optimal moves along the computation. Therefore the choice made in GO is to apply an exhaustive search. GO generates the transformations that reduce the value of N_w to p creating a sequence of sets $scal\_PAT(i)$ containing the PATs obtained using $N_w - i$ processors. The solution must be located in $scal\_PAT(N_w - p)$ where all the PATs are filled and a simple search of the minimum service time is performed to find the best implementation. We have the following assertion:

Assertion 1 (scaling) *The complexity T_{scal} of scaling is bounded by:*

$$T_{scal} \leq (N_{tr} \star N_{sk})^{N_w - p} \star N_{sk}$$

Where N_{tr} is the number of transformations of our system and N_{sk} is the number of skeletons in the program.

Since the transformations reducing the number of processors are: (a) cutting processors from `Farm, Map etc.`, (b) replacing a `Pipe` with `Comp`, (c) collapsing a `Pipe` to a `Seq`, in our system $N_{tr} = 3$.

4.4 Augmenting Parallelism

This phase exploits the available $p - N_w$ processors to minimize the program cost. Since the starting PAT uses the maximum parallelism for the user program, the program must be transformed in order to increase parallelism. Two types of suitable transformations can be applied: (a) the insertion of `Pipe` in place of `Comp` (b) the `Farm` insertion (since the other parallel constructors already use the optimal number of processors). We have the following assertion:

Assertion 2 (Augmenting) *the complexity of augmenting is bounded by:*

$$T_{aug} < (N_{Seq} + N_{Comp})^{(p - N_w)} \star N_{skel}$$

where N_{Comp} and N_{Seq} are the number of `Comp` and `Seq` constructors respectively.

The cost of the augmenting algorithm is furtherly reduced using a pruning technique to decrease the number of generated solutions. In practice the only PATs generated and compared are those in which the number of processors is smaller than p. The pseudo code of the augmenting algorithm is shown in Fig. 6. The procedure `Enumerate` inserts in `solutions` the allocations of processors to constructors satisfying the constraint that the number of processors assigned does not exceed N_{opt}. The Procedure `Prune` eliminates the solutions where the number of processors exceeds p. Finally the procedure `Generate` builds the suitable PATs and `Findopt` selects the solution with the minimum $Tserv$.

5 Case Study

A simple example of the behavior of the GO procedures has been provided using as a case study an image analysis toy-program: IA. IA is a synthetic program

```
int assigned[1..Nseq+Ncomp];
int max[1..Nseq+Ncomp];
int solutions[1..maxsol;1..Nseq+Ncomp];
int Ntr,Nsol;
real Tserv[1..maxsol];
Ntr=Nseq+Ncomp;
for i = 1 to Ntr
    select(Node,i);
    max[i]=max(Nopt(Node.skel,M),p-Nw)
endfor
Enumerate(assigned,max,solutions);
Prune(Solutions,Nsol);
Generate(Solutions,Tserv);
Findopt(Solutions,Tserv,assigned);
```

Fig. 6. The GO augmenting algorithm

including a four stages pipeline where the stages implement: (1) the input from file; (2) the filtering of data; (3) a convolution like computation; (4) the output on file. The sintax tree of IA written in Skel-BSP is shown in Figure 7. The application has been profiled on top of Backus a 10 PCs cluster with PII 266 Mhz CPU and 128 Mbytes RAM running Linux. The machine parameters (M) and the IA *param* list are shown in Tab. 2. Using the values in Tab. 3 the GO produces the transformation visualized in Fig. 8.

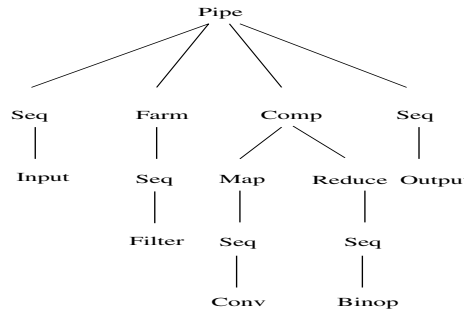


Fig. 7. The Skel-BSP structure of IA

6 Conclusions and Related Work

The paper shows that GO may adapt the structure of a skeletal application to optimize its running time on a specific target architecture. Skel-BSP programs

	g_∞ (μ sec)	L (μsec)	$N_{1/2}$ (bytes)	p (processors)	X
M (Backus)	0.8	1500	64	10	X
	t_{in}	t_{filter}	t_{conv}	t_{red}	t_{out}
T_{seq}(msec)	10	20	80	30	15
	d_{in}	d_{filt}	d_{conv}	d_{red}	d_{out}
sizes (Mbytes)	32	32	32	32	32

Table 2. The parameters of Backus and the IA param list

Skeleton	N_{opt} (proc)	T_{opt} (msec)
Farm	6	8
Map	10	10
Reduce	10	4

Table 3. The values of Nopt and Topt for the IA example on Backus

must be simply recompiled to modify their parallel behavior, therefore Skel-BSP seems a promising approach to reach performance portability. The GO exploit a set of formula providing the costs of optimized implementation of each skeletons based on the EdD-BSP cost model. A related approach is the transformational frame proposed by Gorlatch et al. [7] where an "ad hoc" cost model is proposed to drive a semi-automatic transformation system. The future development of the GO will include:

- evaluating a general approch to efficiently implement the current GO optimization algorithm;
- an extensive validation on several parallel architectures;

finally the set of transformation rules will be enlarged.

References

[1] M. Aldinucci, M. Coppola, and M. Danalutto. Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff. In *Proceedings of International Workshop on Constructive Methods for Parallel Programming*, number MIP-9805 in Technical Report, University of Passau, 1998.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university bsp (pub) library - design, implementation and performance. In *Proceeding of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, Cambridge, Massachusetts, 1989.

[4] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. In D. B. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1994.

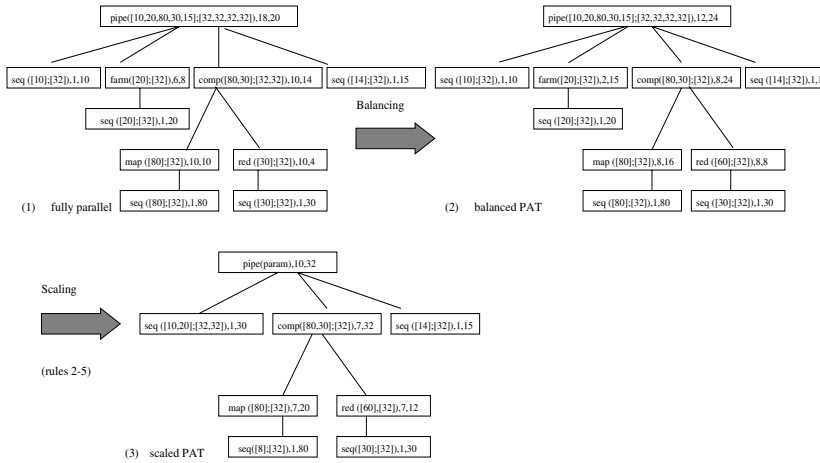


Fig. 8. The optimization of IA

[5] J. Darlington, Y.-K. Guo, Hing Wing To, and J. Yang. Functional skeletons for parallel coordination. *Lecture Notes in Computer Science*, 966:55–67, 1995.

[6] P. De La Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. *Lecture Notes in Computer Science*, 1124:352–360, 1996.

[7] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In Jose Rohlim, editor, *Proc. of Parallel and Distributed Processing. Workshops held in Conjunction with IPPS/SPDP'99*, volume 1586 of *LNCS*, pages 123–137, Berlin, 1999. Springer.

[8] J. M. D. Hill, B. McColl, D. — C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14), 1998.

[9] R. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10-11):1111–1130, December 1991.

[10] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1997.

[11] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[12] A. Zavanella. Optimising Skeletal Stream Parallelism on a BSP Computer. In P. Amestoy, P. Berger, M. Dayde, I. Duff, V. Fraysse, L. Giraud, and D. Ruiz, editors, *Proceedings of EURO-PAR'99*, number 1685 in LNCS, pages 853–857. Springer-Verlag, 1999.

[13] A. Zavanella. Skel-Bsp: Performance Portability for Skeletal Programming. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of HPCN 2000*, volume 1823 of *LNCS*, pages 290–299, 2000.

[14] A. Zavanella. *Skeletons and BSP: Performance Portability for Parallel Programming.* PhD thesis, Dipartimento di Informatica Univesita di Pisa, 2000.

[15] A. Zavanella and S. Pelagatti. Using BSP to Optimize Data-Distribution in Skeleton Programs. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *Proceedings of HPCN99*, volume 1593 of *LNCS*, pages 613–622, 1999.

A Theoretical Framework of Data Parallelism and Its Operational Semantics

Philippe Gerner and Eric Violard

LSIIT-ICPS, Université Louis Pasteur, Strasbourg
{gerner,violard}@icps.u-strasbg.fr

Abstract. We developed a theory in order to address crucial questions of program design methodology. We think that it could unify two concepts of data parallel programming that we consider fundamental as they concern data locality expression: the notions of *alignment* in HPF and *shape* in C*. In this article, we aim at exploring the impact of program transformations on efficiency. For this, we define a formal operational semantics associated with the aforementioned theory.

1 Introduction

The interest of the concepts that lie behind any programming language is that they express the relationship between what the programmer knows about the problem he wants to address and what the compiler knowns about the architecture on which the program will execute. These concepts consist then in an abstract knowledge that the programmer and the compiler share and can use to interact and eventually perform the best implementation.

This is even more true when the architecture is a parallel one, when both the programming and compiling tasks are more complex. The programmer and compiler should influence each other in order to paliate these difficulties. For example, based on these concepts, one could define a language in such a way that a program should be transformed by both entities in a dialog started by the programmer and ended by the compiler. The main point is then to find a good medium, i.e., the concepts that the two entities can handle.

In particular, a major interest of the *data parallelism paradigm* is that it enables the programmer to describe its "parallel" algorithm by choosing a well-known "sequential" one and then focusing on the expression of *data locality*, while the compiler can be left in charge of the distribution onto physical processors, communications management, etc. In some sense, it reveals that the concepts for expressing data locality are of main interest in data parallel programming.

As examples, let us consider standard data parallel programming languages like HPF [1] and C* [6]: they both provide their own way of expressing data locality: *alignment* of arrays for HPF, and *shape* of parallel variables for C*.

In a previous article [7], we proposed a theory which offers a notion of data locality in which those of HPF and C* could join up. The purpose of the article was, to provide a model of this theory, to serve as a formal basis for proving the correctness of data parallel programs.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 668–677, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In the present article, we focus on the execution of data parallel programs, with the objective of comparing the efficiency of the different versions of a program. Estimating the efficiency of a program needs to define its operational semantics.

2 Our Theory

2.1 Objects

Our objects are called *shaped data fields*. A shaped data field is mainly a container of values. Containers without values are sometimes referred to as shapes–in [2], B. Jay defines a very general and abstract notion of shape via a categorical pullback. The originality of our theory consists in its particular notion of shape.

A *shape* is a relation σ between a set of *indices* and a set of *locations*. Both sets are subsets of $\bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$. The indices of a shape serve to access values, and if a value v is accessed through index i, then value v is said to be *located* at all the locations which are related to i by σ. This relation is such that a location is related to at most one index. In Sect. 4, this relation will serve to describe a placement of the values onto a set of virtual processors.

A shaped data field results from the association of a shape with a *data field*. In the literature, indexed collections of values, without locations, are often referred to as data fields (see Alpha [5] and Lisper's formalism [4], as examples). The values of our data fields are indexed by integer tuples.

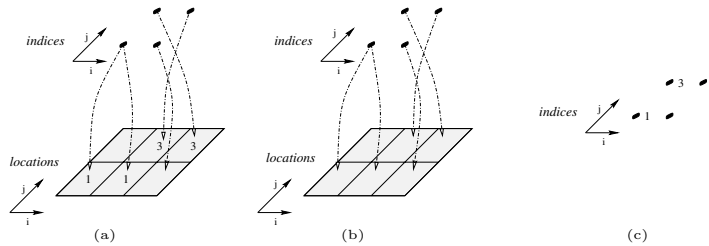


Fig. 1. (a) A shaped data field, (b) its shape and (c) its data field

2.2 Operations

The theory defines three kinds of operations on shaped data fields:

- a *re-indexing* concerns both the shape and the data field of a shaped data field: it changes the indices while keeping the values, and their locations unchanged.

– a *re-locating* does not affect the shape; it changes the locations of some of the values of the data field (with possible duplication or deletion). This change of locations is expressed through a change of indices.

  Any re-indexing or re-locating is determined by a (partial) function from the indices of the new shaped data field to the indices of the old one. For example, Fig. 2(b) shows two shaped data fields: the shaped data field on the right results from a re-locating applied to the shaped data field on the left. This re-locating is determined by the function which associates point of coordinates $(1, j)$ with any point (i, j) in the rectangle $[1..3] \times [1..2]$ of $\mathbb{Z}^2$. Fig. 2(a) illustrates the re-indexing determined by the same function.

– a *global computation* applies the same "scalar" operation on all values. Any classical arithmetical or logical operation on values induces a global computation. In particular, any binary operation on values defines a global computation which combines two shaped data fields having the same shape.

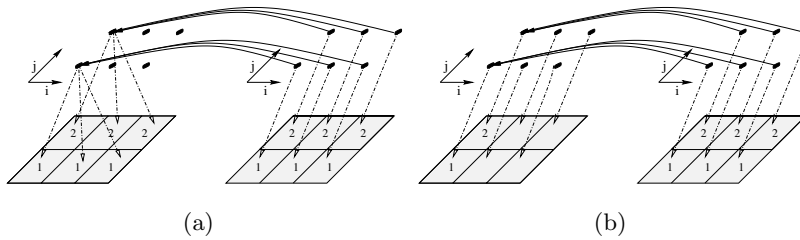


Fig. 2. (a) A re-indexing and (b) a re-locating

2.3 A Minimal Notation Set

Here we introduce a minimal notation set for expressing problems or programs: both are particular cases of what we call *statements*. A *statement* is a finite set of equations whose variables are shaped data fields. Each equation connects two expressions built using the operations on shaped data fields. A variable is an uppercase letter (e.g., A, B, X, ...). The classical arithmetical or logical operations are overloaded with global computations (e.g., $A + B$). We use the following notation for expressing re-indexing and re-locating operations, and also more general cases of dependences: $X.F$, where F is a pair (h, g) of partial functions, is the result of applying first the re-indexing determined by h, and then the re-locating determined by g, to X. When g is the identity, $X.F$ is a re-indexing, and when h is the identity, $X.F$ is a re-locating. In order to denote partial functions, we use the classical lambda-calculus notation $\lambda x.e$, and $f \setminus D$ for the restriction of function f to domain D.

For example, assuming A and B are the shaped data fields on Fig. 2(b), then they satisfy the equation $B = A.Spread$, where *Spread* is pair (h, g), h is the

identity (on D), and g is partial function $\lambda(i,j).(1,j) \setminus D$, with $D = \{\,(i,j) \mid 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\,\}$ (since h is the identity, *Spread* defines a re-locating).

3 Theory Adequacy

This section is intended to link our theory with data locality expression in HPF and in C* and also to show that our theory makes a better compromise than these languages between compiler and programmer.

3.1 Example 1

Let us consider the two following programs in HPF and C*:

```
      REAL A(0:3), B(1:4), C(1:4)

!HPF$ TEMPLATE X(1:8)
!HPF$ ALIGN A(I) WITH X(I+1)
!HPF$ ALIGN B(I) WITH X(2*I)

      ...
      FORALL ( I=1:4 )
        C(I) = A(I-1) + B(I)
      END FORALL
      ...
```

```
shape [9]vector';
real: vector' A',B',C';
...
C' = A' + [.*2]B';
...
```

In the HPF program, template `X` is necessary for aligning arrays `A` and `B` the way we want. The C* program is equivalent to the HPF program to the extent that it expresses a correspondance between array elements and virtual processor that respects the alignment constraints of the HPF program. In the C* program, arrays `A'`, `B'` and `C'` have the same shape, so that for every `i`, value `A'[i]` is on the same virtual processor as values `B'[i]` and `C'[i]`. `A'`, `B'` and `C'` are the result of having the values of `A`, `B`, `C`, placed in such a way that the alignment constraints are respected: for example, `A'[4]` (resp. `B'[4]`) contains the value of element `A[3]` (resp. `B[2]`), so that `A[3]` and `B[2]` are on the same virtual processor.

In the HPF program, the array `C` is not aligned: the placement of values of `C` is left in charge of the compiler. The C* program is more prescriptive as it explicits a particular placement of values and the required communications. In some sense, translating the HPF program to C* reveals the difficulties of the HPF compiler in handling the alignment directives as much as those of the programmer in writing its program in C*.

In our theory, these programs (and their operational meaning) can be expressed by the following statement:

$$C = A.\mathit{Shift} + B.\mathit{Id}$$

where A, B and C are shaped data fields and represent arrays `A`, `B` and `C`, and *Shift*, *Id* are pairs (h, g), (h', g') of partial functions such that $h \circ g$ equals $\lambda(i).(i{-}1) \setminus [1..4]$ and $h' \circ g'$ equals the identity on $[1..4]$. This is enough to express the dependences between the values of arrays `A`, `B` and `C`: the functions $h \circ g$ and $h' \circ g'$ express that, for any $i \in [1..4]$, `C`$[i]$ depends on `A`$[i{-}1]$ and `B`$[i]$. In the HPF program, these dependences are expressed by the `FORALL` block.

Now we precise the functions h, g, h', and g', for expressing the placement used in C* program (which respects the alignment constraints of the HPF program), which is depicted on Fig. 3:

$$\begin{array}{ll} h = \lambda(i).(i{-}1) & h' = g'^{-1} \\ g = id \setminus [1..4] & g' = \lambda(i).(2{\times}i) \setminus [1..4] \end{array}$$

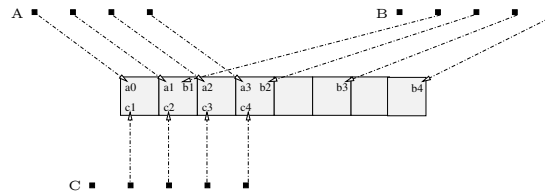


Fig. 3. The placement of values

In comparison with HPF, in our theory a shape is associated with each variable, so that each value of any variable is placed on a set of locations, including the values of variable C in the above statement. And in comparison with C*, the shapes are inferred from the statement, rather than being found by the programmer, as it was necessary for writing the C* program; and also, contrarily to C*, the indices are not names for locations.

3.2 Example 2

Here we discuss the Cooley-Tukey algorithm for the one-dimensional, unordered, radix-2 FFT. Our starting point is the iterative formulation of FFT in [3], p.380. The algorithm is defined by the following set of recurrent equations, where vector $Y[0..n{-}1]$ is the Fourier transform of vector $X[0..n{-}1]$, and $n = 2^r$:

$$R[i,m] = X[i], \qquad \text{for all } (i,m) \in D_0, \quad (1)$$
$$R[i,m] = R[i{\downarrow}m, m{-}1] + (R[i{\uparrow}m, m{-}1]{\times}W[i{\bowtie}m]), \quad \text{for all } (i,m) \in D_{1..r}, \quad (2)$$
$$Y[i] = R[i,m], \qquad \text{for all } (i,m) \in D_r, \quad (3)$$

where R is a temporary $n{\times}r$-matrix, W is a given n-vector that contains the powers of the primitive root of unity in the complex plane (also known as *twiddle factors*), and the following subdomains are considered:

$$D_0 = \{\, (i,m) \mid 0 \le i < n \wedge m = 0 \,\},$$
$$D_{1..r} = \{\, (i,m) \mid 0 \le i < n \wedge 1 \le m \le r \,\},$$
$$D_r = \{\, (i,m) \mid 0 \le i < n \wedge m = r \,\},$$

where "$\downarrow$", "$\uparrow$" and "$\bowtie$" are infixed bitwise operations on the binary representation of index i. For example, $i{\uparrow}m$ is obtained from i by forcing the mth most significant bit to 1. These operations are defined as follows:

$$(b_0 b_1 ... b_{r-1}){\downarrow}m = (b_0 b_1 ... b_{m-1} 0 b_{m+1} ... b_{r-1})$$
$$(b_0 b_1 ... b_{r-1}){\uparrow}m = (b_0 b_1 ... b_{m-1} 1 b_{m+1} ... b_{r-1})$$
$$(b_0 b_1 ... b_{r-1}){\bowtie}m = (b_m b_{m-1} ... b_0 0 ... 0)$$

In order to build a first statement from the above equation set, each array is represented by a shaped data field and each dependence is modeled by a partial function. For example, X and R in equation (1) are considered as shaped data fields, and R depends on X according to the function which maps every (i,m) in D_0 to i, i.e., the partial function $\lambda(i,m).(i) \setminus D_0$. Then, each dependence, say d, is split into two partial functions, say (h,g), such that $d = h \circ g$ in order to define a particular placement of values. This yields the following statement:

$$R.Id_0 = X.Vector_0 \tag{4}$$
$$R.Id_{1..r} = R.Even + (R.Odd \times W.Spread) \tag{5}$$
$$Y.Vector_r = R.Id_r \tag{6}$$

where the definitions of all pairs of functions and dependence functions are reported in Table 1. $R.Id_0$, $R.Id_{1..r}$ and $R.Id_r$ each identifies a piece of R on subdomains D_0, $D_{1..r}$ and D_r, respectively.

Table 1. The definition of a particular placement.

	(h,g)	$(h \circ g)$
Id_0	$= (id,\ id \setminus D_0)$	$id \setminus D_0$
$Id_{1..r}$	$= (id,\ id \setminus D_{1..r})$	$id \setminus D_{1..r}$
Id_r	$= (id,\ id \setminus D_r)$	$id \setminus D_r$
$Vector_0$	$= (\lambda(i,m).(i) \setminus D_0,\ id)$	$\lambda(i,m).(i) \setminus D_0$
$Even$	$= (id,\ \lambda(i,m).(i{\downarrow}m, m-1) \setminus D_{1..r})$	$\lambda(i,m).(i{\downarrow}m, m-1) \setminus D_{1..r}$
Odd	$= (id,\ \lambda(i,m).(i{\uparrow}m, m-1) \setminus D_{1..r})$	$\lambda(i,m).(i{\uparrow}m, m-1) \setminus D_{1..r}$
$Spread$	$= (\lambda(i,m).(i{\bowtie}m) \setminus D_{1..r},\ id)$	$\lambda(i,m).(i{\bowtie}m) \setminus D_{1..r}$
$Vector_r$	$= (\lambda(i,m).(i) \setminus D_r,\ id)$	$\lambda(i,m).(i) \setminus D_r$

By this statement, vector X (resp. Y) is aligned with the first (resp. last) column of matrix R. Moreover, the twiddle factors have been replicated onto the matrix so that the dependence defined by *Spread* does not involve any communication. This statement describes the *binary-exchange algorithm* if the indices of variable R are identified with locations. In the following, R will be referred to as the *template* of this statement.

4 Operational Semantics

All the statements we have seen in the previous section are computable. They are examples of what we call *well-formed statements*. This class of statements is associated with a proof system [7] and an operational semantics.

4.1 Well-Formed Statements

Informally, any well-formed statement is such that the shape of each variable can be inferred uniquely from the shape of one of them. Moreover, we restrict ourselves to statements in single-assignment form.

Definition 1. *A* well-formed statement*, is a statement, say P:*

- *with some variables declared as* input variables*, other ones declared as* output variables*, and with a single one declared as the* template *of the statement (and referred to as T);*
- *whose every equation has the form*

$$X_0.(h_0, g_0) = \phi(X_1.(h_1, g_1), \ldots, X_n.(h_n, g_n)) \ ,$$

  where ϕ is a global computation, and for all $k \in [1..n]$, X_k is a variable of P, h_k and g_k are partial functions from I to I, where $I = \bigcup_{n \in \mathbb{N}} \mathbb{Z}^n$, $g_0 = id|_D$, where id is the identity on I, and either $h_0 = id$ (case 1), or $h_0 \neq id$, and $\forall k \in [1..n], h_k = id$ (case 2);
- *such that, the following set of equations, denoted $S(P)$, has a unique solution for any σ_T. Its variables are partial functions from I to I, and we have one variable σ_X for each variable X of P. $S(P)$ is the set of all the equations $\sigma_{X_k} = h_k \circ \sigma_{X_0}$, $k \in 1..n$, induced from every equation of P of case 1, and equations $\sigma_{X_0} = h_0 \circ \sigma_{X_k}$ induced from equations of case 2;*
- *and such that, if a variable X appears in more that one left-hand side, in terms $X.(h_1, id|_{D_1}), \ldots, X.(h_q, id|_{D_q})$, then $\forall k, k' \in 1..q$, $k \neq k' \Rightarrow D_k \cap D_{k'} = \emptyset$.*

Relative Placement of Variables. The system $S(P)$ associated with a well-formed statement P defines the placement of the values of each shaped data field on a set of *locations*. For each variable X, this placement is defined by σ_X the following way: the values of X accessed by index i are placed at locations ${\sigma_X}^{-1}(i)$ (in the sequel, this set is denoted $\mathrm{loc}(X, i)$).

$S(P)$ defines the functions σ_X as follows. The set of locations is I, and the values of template variable T are considered placed on I with $\sigma_T = id$, so that at most one value is accessed through index i, and this value is then located at location $l = i$. The placement of all the other variables are uniquely inferred from this one.

4.2 States and Transitions

A *state* is a set of *stores*. We have one store for each location. The store associated with location $l \in I$ is a partial function $\rho_l : Var \times I \rightharpoonup V$, where Var is the set of variables of statement P, and V is a set of values. Intuitively, $\rho_l(X, i)$ is the current value of X, indexed (in the index set of X) by i, and placed at location l. A *transition* describes a change in the store of a location. So, a transition induces a change from one state to another. The transitions can occur concurrently.

The execution begins with an *initial state*. In the initial state, if X is an input variable, and $\rho_l(X, i)$ is defined, then $l \in \mathrm{loc}(X, i)$, and $\forall l' \in \mathrm{loc}(X, i), \rho_{l'}(X, i) = \rho_l(X, i)$; and if X is not an input variable, then $\forall l \in I, \forall i \in I$, $\rho_l(X, i)$ is undefined. From the initial state, the transitions apply, in a non-deterministic way, and concurrently, until no more transition can apply.

Transitions. For every equation $X_0.(h_0, g_0) = \phi(X_1.(h_1, g_1), \ldots, X_n.(h_n, g_n))$, of P, we have the following transition rule, where $i \in I$ is any index, $l_0, l_1, \ldots, l_n \in I$ are any locations, and d_k is for $h_k \circ g_k$, $k \in 0..n$:

$$\frac{\begin{array}{c}\mathrm{UNDEF}_0 \wedge l_0 \in \mathrm{loc}(X_0, d_0(i)) \\ \forall k \in 1..n, \mathrm{DEF}_k \wedge (l_0 = l_k \vee l_0 \not\in \mathrm{loc}(X_k, d_k(i)))\end{array}}{\rho_{l_0} \longrightarrow \rho_{l_0}[(X_0, d_0(i)) \leftarrow \phi(\rho_{l_1}(X_1, d_1(i)), \ldots, \rho_{l_n}(X_n, d_n(i)))]} \tag{7}$$

where $\rho[(X, i) \leftarrow v]$ is the store ρ except that it maps (X, i) to value v, UNDEF_0 means that $d_0(i)$ is defined and $\rho_{l_0}(X_0, d_0(i))$ is undefined, and DEF_k $(k \in 1..n)$ means that $d_k(i)$ and $\rho_{l_k}(X_k, d_k(i))$ are defined.

Computations: Assuming the Owner-Computes Rule, it is to be understood that, in the application of a transition, the computation $\phi(\ldots)$ is made at location l_0. Note that, when $\mathrm{loc}(X_0, d_0(i))$ contains more than one location, the calculus will be made several times. Such a case is useful when duplicating the calculus avoids some communications.

Communications: When $l_0 \neq l_k$, there is a *communication* for sending the value from location l_k, to location l_0, because the computation on l_0 needs this value. The transition does not apply if $l_0 \neq l_k \wedge l_0 \in \mathrm{loc}(X_k, d_k(i))$, because in this case the value at location l_k will be available locally, that is, at location l_0.

5 Example

Let us consider again the FFT example in Sect. 3.2, with $n = 2^4$. Suppose we are given 4 physical processors. Equation (5) is the core of the algorithm. R is the template of the statement, so $\sigma_R = id$, which means that if R contains a value at index (i, m), then this value lies on location (i, m). So, the transition rule associated to this equation breaks down to the following rule:

$$\frac{\begin{array}{c}(i,m) \in D_{1..r} \ \wedge \ \rho_{i,m}(R,(i,m)) \text{ is undefined} \\ \wedge \ \rho_{i\downarrow m,m-1}(R,(i\downarrow m,m-1)) \text{ is defined} \\ \wedge \ \rho_{i\uparrow m,m-1}(R,(i\uparrow m,m-1)) \text{ is defined}\end{array}}{\begin{array}{l}\rho_{i,m} \longrightarrow \rho_{i,m}[(R,(i,m)) \leftarrow \rho_{i\downarrow m,m-1}(R,(i\downarrow m,m-1)) \\ \qquad\qquad + (\rho_{i\uparrow m,m-1}(R,(i\uparrow m,m-1)) \times \rho_{i,m}(W, i\bowtie m))]\end{array}} \quad (8)$$

This rule is the only rule, among those induced by the statement, that generates communications. The communications come from the fact that indices (i,m), $(i\downarrow m, m-1)$ and $(i\uparrow m, m-1)$ represent different locations (because R is the template), so that for doing the computation at (i,m), 2 communications are needed for getting the required values which sit at locations $(i\downarrow m, m-1)$ and $(i\uparrow m, m-1)$. We get a total of $2 \times \text{card}(D_{1..r}) = 2 \times n \times r = 2 \times 16 \times 4 = 128$ communications.

The rule also shows that the axis of index m follows the direction of time during execution of the program. So, an obvious projection onto physical processors is along this axis. For each (i,m), either $i\downarrow m = i$ or $i\uparrow m = i$, so that this projection reduces the number of communications by a half, to 64. But we have only 4 processors, not 16, and in this case what is classically done is, distributing 4 contiguous elements of the array (on the first dimension of R) onto one processor. The number of communications is then reduced further, from 64 to $64 - (2 \times 16) = 32$ communications (because the dependences in the last 2 steps require no communications).

A different algorithm could be described, characterized by another placement of values onto the processors: the two-dimensional transpose algorithm. This new algorithm can be expressed by the same statement as in Sect. 3.2, except that we re-index variable R, obtaining a new template variable R'. This is expressed by just adding to the statement the following equation:

$$R.(\mathit{transf}, \mathit{id}) = R' \ , \quad (9)$$

where $\mathit{transf}(i,j,m) = \begin{cases} (j+i\times\sqrt{n}, m), \text{ if } 0\leq i,j<\sqrt{n} \wedge 0\leq m<r/2 \\ (i+j\times\sqrt{n}, m), \text{ if } 0\leq i,j<\sqrt{n} \wedge r/2\leq m<r \end{cases}$

This transformation is sketched on Fig. 4.

Since we have only changed the template, the number of (virtual) communications is the same for this new algorithm. The interpretation of the new template is the following one: the third axis of R' corresponds to time, and the first two axes give a square of locations of dimensions $\sqrt{n} \times \sqrt{n}$. Again, an obvious projection onto processors is along the third axis.

The idea of the transpose algorithm is, if given 4 processors, we distribute each column onto a processor for the first $r/2 = 4/2 = 2$ iterations, and each line onto a processor for the last $r/2 = 2$ iterations: then, with the above projection, no communication is required for the first 2 iterations, nor for the last 2 iterations. But a *re-distribution* generating communications, which corresponds to the transposition of a matrix of values, is necessary between these two big steps of

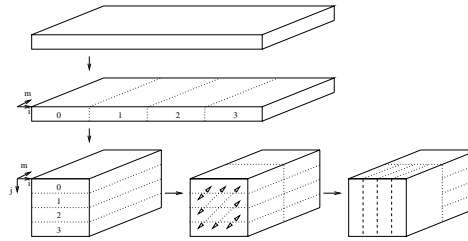


Fig. 4. A sketch of the transformation in the case n=16: the domain of R is first rearranged as a parallelepipede and then divided into two parts; last step consists in transposing the matrices forming the first of these two parts.

the algorithm. So, the total number of communications is the number of communications required for the transposition, which is $(\sqrt{n} \times \sqrt{n}) - \sqrt{n} = 16 - 4 = 12$.

6 Conclusion

In our opinion, in reference to automatic parallelization, the programmer must provide some additional information to the compiler in order that the compiler can reach more efficiency. In particular, data locality expression is of great importance in data parallelism. Therefore, we proposed a theoretical framework which unify the two main concepts for expressing data locality: *alignment* of arrays in HPF and *shape* declaration in C*.

This framework allows the programmer and the compiler to share a minimum knowledge to reach efficiency. We think that it could help both the programmer and the compiler to transform the program in order to reach the best implementation.

References

[1] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.
[2] C.B. Jay. A semantics for shape. *Sc. of Computer Programming*, 25:251–283, 1995.
[3] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.
[4] B. Lisper. *Data Parallelism and Functional Programming.* LNCS 1132-Tutorial Series, 1996.
[5] C. Mauras. Alpha *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.
[6] Thinking Machines Corp. *C* Programming Guide*, November 1990.
[7] E. Violard. What really is data parallelism? Technical Report RR 00-01, ICPS, Université Louis Pasteur, January 2000.

A Pattern Language for Parallel Application Programs*

Berna L. Massingill[1], Timothy G. Mattson[2], and Beverly A. Sanders[3]

[1] Department of Computer and Information Science and Engineering
University of Florida; Gainesville, FL
blm@cise.ufl.edu†

[2] Parallel Algorithms Laboratory, Intel Corporation
timothy.g.mattson@intel.com

[3] Department of Computer and Information Science and Engineering
University of Florida; Gainesville, FL
sanders@cise.ufl.edu

Abstract A design pattern is a description of a high-quality solution to a frequently occurring problem in some domain. A pattern language is a collection of design patterns that are carefully organized to embody a design methodology. A designer is led through the pattern language, at each step choosing an appropriate pattern, until the final design is obtained in terms of a web of patterns. This paper describes a pattern language for parallel application programs. The goal of our pattern language is to lower the barrier to parallel programming by guiding a programmer through the entire process of developing a parallel program.

1 Introduction

Parallel hardware has been available for decades, and is becoming increasingly mainstream. Parallel software that fully exploits the hardware is much rarer, however, and mostly limited to the specialized area of supercomputing. We believe that part of the reason for this is that most parallel programming environments, which focus on the implementation of concurrency rather than higher-level design issues, are simply too difficult for most programmers to risk using them.

A *design pattern* describes, in a prescribed format, a high-quality solution to a frequently occurring problem in some domain. The format is chosen to make it easy for the reader to quickly understand both the problem and the proposed solution. Because the pattern has a name, a collection of patterns provides a vocabulary with which to talk about the these solutions.

A *pattern language* is a structured collection of design patterns, with the structure chosen to lead the user through the collection of patterns in such a

* We acknowledge the support of Intel Corporation, the National Science Foundation, and the Air Force Office of Scientific Research.

† Current address: Department of Computer Science, Trinity University, San Antonio, TX; bmassing@trinity.edu.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 678–681, 2000.
© Springer-Verlag Berlin Heidelberg 2000

way that complex systems can be designed using the patterns. At each decision point, the designer selects an appropriate pattern. Each pattern leads to other patterns, resulting in a final design in terms of a web of patterns. Thus a pattern language embodies a design methodology and provides domain-specific advice to the application designer. (In spite of the overlapping terminology, a pattern language is *not* a programming language.) The first pattern language was elaborated by Alexander [1] and addressed problems from architecture, landscaping, and city planning. More recently, design patterns were introduced to the software engineering community by Gamma, Helm, Johnson, and Vlissides [3] via a collection of design patterns for object-oriented programming.

This paper describes a pattern language for parallel application programs. The goal is to lower the barrier to parallel programming by guiding a programmer through the entire process of developing a parallel program. The target audience is experienced programmers who may lack experience with parallel programming. The programmer brings to the process a good understanding of the actual problem to be solved and works through the pattern language to obtain a detailed parallel design or perhaps working code. The current state of the whole pattern language may be seen at our Web site [7]; the patterns are extensively hyperlinked, allowing the programmer to work through the pattern language by following links. In this paper, space constraints allow only an overview of the pattern language and a brief discussion of related approaches; see [7] and [8] for complete patterns and examples of their use.

2 Organization of the Pattern Language

The pattern language is organized into four design spaces — FindingConcurrency, AlgorithmStructure, SupportingStructures, and ImplementationMechanisms — which form a linear hierarchy, with FindingConcurrency at the top.

FindingConcurrency. This design space is concerned with structuring the problem to expose exploitable concurrency; the programmer working at this level focuses on high-level algorithmic issues and reasons about the problem to expose potential concurrency. The space includes four major patterns: **GettingStarted** helps the programmer understand where to start in solving a problem using a parallel algorithm. **DecompositionStrategy** helps the programmer decide whether the problem should be decomposed based on a data decomposition, a task decomposition, or a combination of the two. **DependencyAnalysis** helps the programmer understand how the elements of the decomposed problem depend on each other. Finally, **Recap** helps the programmer bring together the results from the other patterns and prepare to work in the AlgorithmStructure design space.

AlgorithmStructure. This design space is concerned with structuring the algorithm to take advantage of the potential concurrency exposed in the previous level. Patterns in this space describe overall strategies for exploiting concurrency.

They can be divided into the following four groups, plus **ChooseStructure**, which helps the programmer select an appropriate pattern from those in this space.

"Organize by ordering" patterns. These patterns are used when the ordering of groups of tasks is the major organizing principle for the parallel algorithm. An example is **PipelineProcessing**, which decomposes the problem into ordered groups of tasks connected by data dependencies.

"Organize by tasks" patterns. These patterns are those for which the tasks themselves are the best organizing principle ("task parallelism"). Two examples are **EmbarrassinglyParallel**, which decomposes the problem into a set of independent tasks that can be executed concurrently (examples are algorithms based on task queues and random sampling); and **DivideAndConquer**, which solves the problem by recursively dividing it into subproblems, solving the subproblems independently, and then combining the subsolutions.

"Organize by data" patterns. These patterns are those for which the decomposition of the data is the major organizing principle in understanding the concurrency ("data parallelism"). An example is **GeometricDecomposition**, which decomposes the problem space into discrete subspaces and solves the problem by computing solutions for the subspaces, with the solution for each subspace typically requiring data from a small number of other subspaces (examples include grid-based computations).

SupportingStructures. This design space represents an intermediate stage between the problem-oriented AlgorithmStructure patterns and the machine-oriented ImplementationMechanisms "patterns"; it includes both patterns that represent program-structuring constructs (e.g., **ForkJoin**) and patterns that represent commonly used shared data structures (e.g., **SharedQueue**). Ideally patterns in this space would be implemented as part of a library.

ImplementationMechanisms. This design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide pattern-based descriptions of common mechanisms (e.g., barriers and message-passing) for managing and coordinating processes or threads. Patterns in this design space, like those in the SupportingStructures space, describe entities that strictly speaking are not patterns at all, but which we include in our pattern language to provide a complete path from problem description to code.

3 Related Work

Design patterns and pattern languages were first introduced into software engineering in [3]. Early work on patterns dealt mostly with object-oriented sequential programming, but more recent work [10, 4, 9] addresses concurrent programming as well. Algorithmic skeletons and frameworks capture very high-level patterns that provide the overall program organization, with the user providing

lower-level code specific to an application. Skeletons, as in [2], are typically envisioned as higher-order functions, while frameworks often use object-oriented technology. Particularly interesting is recent work [5] on using design patterns to generate frameworks for parallel programming from pattern template specifications. Programming archetypes [6] combine elements of all of the above categories, capturing common computational and structural elements at a high level and also providing a basis for implementations including both frameworks and code libraries, but do not directly address the question of how to choose an appropriate archetype for a particular problem.

4 Conclusions

We have described a pattern language for parallel programming. Currently, the top two design spaces (FindingConcurrency and AlgorithmStructure) are relatively mature, with several of the patterns having undergone scrutiny at a writers' workshop for design patterns [8]. The lower two design spaces are still under construction, but the pattern language is usable, and several case studies are in progress (see [7]). Preliminary results of the case studies and feedback from the writers' workshop leave us optimistic that our pattern language can indeed achieve our goal of lowering the barrier to parallel programming.

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[2] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1997.

[5] S. MacDonald, D. Szafron, J. Schaeffer, and S. Bromling. From patterns to frameworks to parallel programs, 1999. Submitted to *IEEE Concurrency*, August 1999; see also `http://www.cs.ualberta.ca/~stevem/papers/IEEECON99.ps.gz`.

[6] Berna L. Massingill and K. Mani Chandy. Parallel program archetypes. In *Proceedings of the 13th International Parallel Processing Symposium (IPPS'99)*, 1999.

[7] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A pattern language for parallel application programming. Available at `http://www.cise.ufl.edu/research/ParallelPatterns`.

[8] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for parallel application programs. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99)*, 1999.

[9] Jorge Ortega-Arjona and Graham Roberts. Architectural patterns for parallel programming. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*, 1998.

[10] D. C. Schmidt. The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software. `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`, 1993.

Oblivious BSP

Jesus A. Gonzalez [1], Coromoto Leon [1], Fabiana Piccoli [2], Marcela Printista[2], Jos□L. Roda [1], Casiano Rodriguez [1], and Francisco de Sande[1]

[1]Dpto. Estadstica, Investigacin□Operativa y Computacin□,
Universidad de La Laguna, Tenerife, Spain
casiano@ull.es
[2] Universidad Nacional de San Luis
Ej□rcito de los Andes 950, San Luis, Argentina
mprinti@unsl.edu.ar

Abstract. The BSP model can be extended with a zero cost synchronization mechanism, which can be used when the number of messages due to receive is known. This mechanism, usually known as "oblivious synchronization" implies that different processors can be in different supersteps at the same time. An unwanted consequence of these software improvements is a loss of prediction accuracy. This paper proposes an extension of the BSP complexity model to deal with oblivious barriers and shows its accuracy.

1 Introduction

The asynchronous nature of some parallel paradigms like farms and pipelines hampers the efficient implementation in the scope of a flat-data-parallel global-barrier Bulk Synchronous Programming (BSP [4]) software like the BSPLib [3]. To overcome these limitations, the Paderborn University BSP library (PUB) offers the use of collective operations, processor-partition operations and oblivious synchronizations [1]. In addition to the BSP most common features, PUB provides the capacity to partition the current BSP machine into several subsets, each of which acts as an autonomous BSP computer with their own processor numbering and synchronization points. One of the most novel features of PUB is oblivious synchronization. It is implemented through the *bsp_oblsync(bsp,n)* function, which does not return until *n* messages have been received. Although its use mitigates the synchronization overhead, it implies that different processors can be in different supersteps at the same time. The BSP semantic is preserved in PUB by numbering the supersteps and by ensuring that the receive thread buffers messages that arrive out of order until the correct superstep is reached. An unwanted consequence of these software improvements is a loss of accuracy when using the BSP complexity model to predict the running time.

Runtime systems oriented to the flat BSP model try to bring actual machines closer to the BSP ideal machine by packing individual messages generated during a

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 682-685, 2000.
© Springer-Verlag Berlin Heidelberg 2000

superstep and optimizing communication time by rearranging the order in which messages are sent at the end of the superstep [3]. This policy reduces the influence of the communication pattern, since it gives place to an *AllToAll* communication pattern at the end of each superstep. According to this, the actual overlapping of supersteps produced by PUB machine partitioning and oblivious synchronization makes unfeasible the former implementation approach and may lead to congestion (hot spots) and therefore to a wider variability in the observed bandwidth.

In the next section we propose an extension of the BSP model for PUB parallel programs: the **O**blivious **BSP** model. The model intends to cover the complete class of PUB programs. The computational times on a CRAY T3E of a PUB Divide and Conquer algorithm were predicted with an error percentage under 3%.

2 The Oblivious BSP Model

The execution of a PUB program on a BSP machine $X=\{0,1,...,p-1\}$ consists of supersteps. As a consequence of the oblivious synchronizations, processors may be in different supersteps at a given time. However, still is true that:

- Supersteps can be numbered starting at 1.
- The total number of supersteps R, performed by all the p processors is the same.
- Although messages sent by a processor in superstep s may arrive at another processor executing an earlier superstep r, communications are made effective only when the receiver processor reaches the end of superstep s.

Lets assume in first instance that no processor partitioning is performed in the analyzed task T. If the superstep s ends in an oblivious synchronization, we define the set $\Omega_{s,i}$ for a given processor i and superstep s as the set

$$\Omega_{s,i} = \{ j \in X / \text{Processor } j \text{ sends a message to processor } i \text{ in superstep } s \} \cup \{ i \} \qquad (1)$$

while $\Omega_{s,i} = X$ when the superstep ends in a global barrier synchronization. In fact, this last expression can be considered a particular case of formula (1) if it is accepted that a barrier synchronization carries an all-to-all communication. Processors in the set $\Omega_{s,i}$ are called "**the incoming partners of processor *i* in step *s***". Usually it is accepted that all the processors start the computation at the same time. The presence of partition functions in PUB forces us to consider the most general case in which each processor i joins the computation at a different initial time ξ_i. The **Oblivious BSP** time $\Phi_{s,i}(T, X, \xi)$ when processor $i \in X$ executing task T finishes superstep s is recursively defined by the formulas:

$$\begin{aligned} \Phi_{1i} &= max \{ w_{1,j} + \xi_i / j \in \Omega_{1,i} \} + (g\, h_{1,i} + L_b), \;\; i = 0,1,..., p-1 \\ \Phi_{s,i} &= max\{\Phi_{s-1,j} + w_{s,j} \;/\; j \in \Omega_{s,i} \} + (g\, h_{s,i} + L_b), \; s=2,..,R, \;\; i=0,1,..., p-1 \end{aligned} \qquad (2)$$

where $w_{s,i}$ and $h_{s,i}$ respectively denote the time spent in computing and the number of packet units communicated by processor i in step s:

$$h_{s,i} = max \{ in_{s,j} @ out_{s,j} / j \in \Omega_{s,i} \}, \;\; s = 1,...,R, \;\; i = 0,1,..., p-1 \qquad (3)$$

and $in_{s,i}$ and $out_{s,i}$ respectively denote the number of packet units incoming/outgoing to/from processor i in the superstep s. The @ operation is defined as *max* or the *sum* depending on the input/output capabilities of the network interface. While the gap g has the same meaning that in BSP, the value L_b is related with the start-up latency. This oblivious latency value L_b is different from the barrier synchronization latency value L used in global synchronization superteps. Another issue to discuss is what is the appropriate size of a unit communication packet. This size depends on the target architecture. Rather than exhibiting linear behavior, the communication time takes the form of a piecewise linear function. There is always a slope for small size messages different from the slope for large messages. We define the *OBSP Packet Size* as the size in which the time/h-relation size curve of the architecture has the first inflection point. Special gap g_0 and latency L_0 values have to be used for messages of sizes smaller than the *OBSP unit packet size*.

The formula (2) says that processor i in its step s can not start the reception of its messages before it has finished its computing time $\Phi_{s-1,i} + w_{s,i}$ and so have done the other processors j sending messages to processor i. The formulas charge the communication time of processor i with the maximum communication time of any of its incoming partner processors.

The times of a PUB task T in OBSP are given by the vector of times $\Phi_{R,j}\ (T, X, \xi)$ when processor $j=0,...,p-1$ finishes task T. R is the total number of supersteps, X is the set of processors executing the task T, and $\xi = (\xi_0, ..., \xi_{p-1})$ is the vector of starting times. At any time, the processors are organized in a *hierarchy of processor sets*. A processor set in PUB is represented by a structure called a BSP object. Let be Q a set of processors (i.e. a BSP object) executing a task T. When all the processors in Q call the function *bsp_partition* with arguments *(t_bsp* Q, t_bsp* S, int r, int* Y)* the set Q is divided in r disjoint subsets S_i:

$$Q = \cup_{0 \le j \le r-1} S_j\ ;\ S_i = \{Y[i-1], ..., Y[i]-1\}\quad 1 \le i \le r-2 \text{ and } S_0 = \{0, ..., Y[0]-1\}$$

After the partition step each subgroup S_i acts as an autonomous BSP computer with its own processor number, message queue and synchronization points. The time when processor $j \in S_i$ finishes the task T_i executed by the object BSP S_i is given by

$$\Phi_{Ri,j}\ (T_i, S_i, \Phi_{s-1,j} + w^*_{s,j}) \text{ such that } j \in S_i\ i=0, ..., r-1 \qquad \textbf{(4)}$$

where R_i is the number of supersteps performed in task T_i and $w^*_{s,j}$ is the computing time performed by processor j before its call to the *bsp_partition* function and s is the current superstep number.

Observe that subgroups are created in a stack-like order. The two functions *bsp_partition* and *bsp_done* incurs no communication. This implies that different processors in a given subset can arrive at the partition process (and leave it) at different times. From the point of view of the parent submachine, the code executed between the call to *bsp_partition* and *bsp_done* behaves as computation (i.e. like a call to a subroutine).

Another essential difference of the Oblivious Model proposed here with the BSP model is the way the computing time W is carried out. Instead of using a single parameter s to characterize the computational power, our proposal is to return to an approach nearer to the original RAM model: a sequential algorithm A with input of

size n has complexity of order $O(f(n))$ if and only if there are constants C and D such that the time $T_{A,M}(n)$ invested by a computer M executing A on such input holds the equation $T_{A,M}(n) \leq C*f(n)+D$. The values of constants C and D can be estimated from the actual times $T_{A,M}(n)$ of the algorithm A on the worst (or average) cases of size n.

Table 1 presents the prediction accuracy of the OBSP for a nested parallel recursive Discrete Fast Fourier Transform PUB algorithm [2].

Table 1. Real and OBSP predicted times for the FFT algorithm in the CRAY T3E.

Processors	**2**	**4**	**8**	**16**
Real	6.8967	3.7712	2.2552	1.5385
Predicted OBSP	6.8001	3.6838	2.2118	1.5226
% Error	1.40	2.32	1.92	1.03

Acknowledgements

We would like to thank to Centre de Computaci i Communicacions Catalunya, Edinburgh Parallel Computing Centre, Centro de Investigaciones Energ ticas, Medioambientales y Tecnol gicas, Universidad de La Laguna and Universidad Nacional de San Luis. This research has been partially supported by Comisi n Interministerial de Ciencia y Tecnologa under project TIC1999-0754-C03.

References

1. Bonorden O. Juurlink B. von Otte I. Rieping I. *The Paderborn University BSP (PUB) Library - Design Implementation and Performance*. Technical Report TR-RSFB-98-063. Heinz Nixdorf Institute. Paderborn University. 1998.
2. Gonzalez J.A. , Leon C., Piccoli F., Printista M., Roda J.L., Rodriguez C., Sande F. *Oblivious BSP*. Internal Report. http://nereida.deioc.ull.es/html/obsp.ps.gz
3. Hill J. McColl W. Stefanescu D. Goudreau M.. Lang K. Rao S. Suel T. Tsantilas T. Bisseling R. *BSPLib: The BSP Programming Library*. Parallel Computing. 24(14) pp. 1947-1980. 1988.
4. Valiant L.G. *A Bridging Model for Parallel Computation*. Communications of the ACM. 33(8). pp. 103-111. 1990.

A Software Architecture for HPC Grid Applications

Steven Newhouse, Anthony Mayer, and John Darlington

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London,
SW7 2AZ,
UK
sjn5,aem3,jd@doc.ic.ac.uk
http://hpc.doc.ic.ac.uk/environments/

Abstract. We introduce a component software architecture designed for demanding grid computing environments that allows the optimal performance of the assembled component applications to be achieved. Performance over the assembled component application is maintained through inter-component static and dynamic optimisation techniques. Having defined an application through both its component task and data flow graphs we are able to use the associated performance models to support application level scheduling. By building grid aware applications through reusable interchangeable software components with integrated performance models we enable the automatic and optimal partitioning of an application across distributed computational resources.

1 Introduction

The emergence of local, national and international high-bandwidth networking allows physcially distributed computing hardware resources to be linked enabling the development of 'computational grids'. These emerging grids present new challenges in automatic scheduling, application partitioning and resource management to deliver an effective computing environment. Effectively exploiting these dynamic and heterogeneous networking and storage resources requires automatic data partitioning to enable automatic run-time scheduling. Such scheduling only becomes feasible once performance information is integrated into the application.

We introduce a scientific component framework for HPC applications which provides relevant abstractions to the end-user, scientist and numerical programmer. Application performance is maintained through static and dynamic component optimisations which allow component implementations to be matched to the execution architecture. By adopting object and component based programming techniques we can deliver the functionality of skeletons through a conventional abstraction mechanism.

We present a simple example to illustrate these concepts and refer the reader to our complete paper (http://www-icpc.doc.ic.ac.uk/components/papers).

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 686–689, 2000.
© Springer-Verlag Berlin Heidelberg 2000

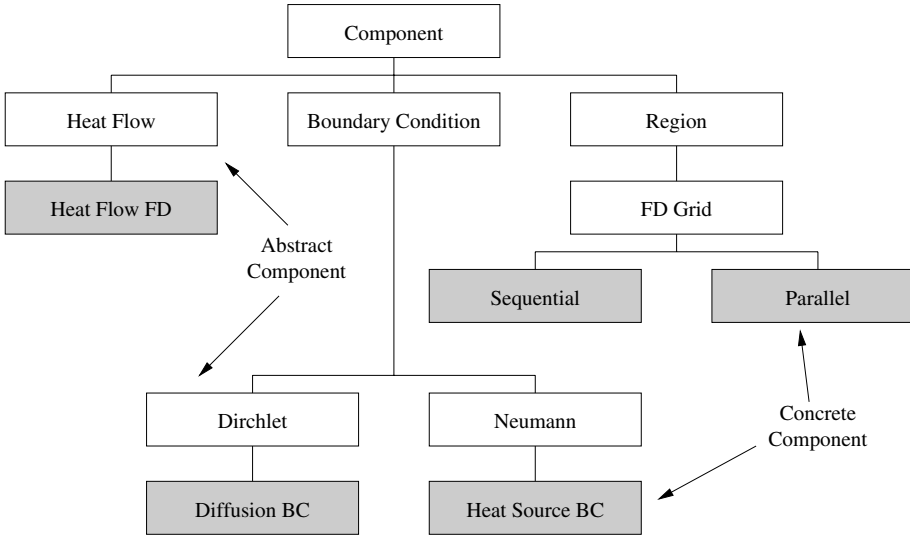


Fig. 1. Component Repository for a simple Finite Difference Problem showing abstract (white boxes) and concrete (grey boxes) components.

2 An Example: Heat Flow in an Insulated Bar

We will illustrate this software architecture with a simple scientific example of heat flow in an insulated beam. A simplified component repository is illustrated in Figure 1 to demonstrate the implementation hierarchy and the use of abstract and concrete components.

1. **Problem Definition.** The problem is defined within a graphical PSE linked to a component repository (Figure 2). A component will have context dependent representations. It may be a code segment in an execution context, have a visual representation when being visually composed to build an application or have a three dimensional representation within the PSE.
   In the example, the Region component is selected from the repository, placed within the graphical PSE and manipulated to define the physical domain of the bar. Properties such as the boundary and initial conditions are attached to the edges and domain to characterise the behaviour of the Region.
2. **The Component Network.** The problem defined within the graphical PSE can also be expressed as a component network (Figure 3). This allows further non-graphical components to be added into the component assembly to complete the application which will be used to solve the problem. The parameters within the components can be adjusted to further define the physical problem.
3. **Valid Implementation Options.** The component network which has been defined by the user is validated for its correctness. In the heat flow example

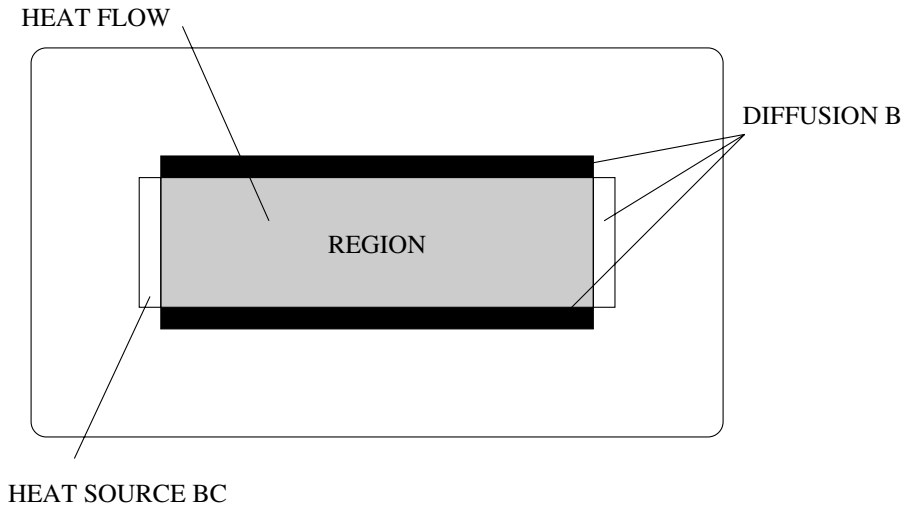


Fig. 2. A graphical Problem Solving Environment being used to physically define the analysis by manipulating graphical components.

a boundary condition must be applied to each edge of the physical Region and one or more properties may be applied to the whole Region to give it some physical characteristics (eg. heat flow, elastic material, etc). These conditions are essential pre-requisites for any valid component network and can be incorporated at a low-level into the Region component.

The validated component network is compared to the available components in the repository. The static optimisation process described earlier is used to examine all of the feasible implementation options. In this simple example a choice has to be made between solving the problem using a sequential or parallel approach.

4. **Scheduling.** The valid implementation options are passed to the scheduler to match the application to a computational resource. This process uses the performance models derived from the assembled application and the performance of the computational resources to determine, within the constraints specified by both the user and resource provider, the resources needed to execute the application. In this example, the application's overall performance model, the target architectures and the problem size will show when it is appropriate to move from a serial to a parallel implementation. Further examination of the parallel performance model will yield the ideal domain decomposition for this problem size.

   The effectiveness of cross platform scheduling can also be assessed as the component assembly will provide a profile of the computational requirements over time. If a matrix is being generated and then solved, it may be quicker to generate the matrix on a scalar parallel machine and transfer the data

to a vector machine for the solution phase rather than execute both tasks on the vector machine. The data flow and execution tasks graphs within the application, developed during the component composition process allow computing and networking performance models to be used to assess these alternative implementation options.

5. **Execution.** The component assembly is finally passed to the resources for execution. The execution and data flow graphs can still be exploited to examine the benefits of migrating the application to faster resources if they become available or the benefits of adding additional resources to the current calculation.

3 Conclusions

The proposed architecture, which we are developing and implementing across several applied scientific domains, is an extension of current or established research in compositional programming techniques. The skeletons work has already demonstrated the validity of using performance models to guide implementation and data layout decisions. We are able to exploit standard component models, such as Java Beans and CORBA, to contain the skeleton code fragments.

The repository will contain a variety of software components presenting usable abstractions to the end-user, the scientist and the numerical programmer. By defining standard interfaces, software will become easier to reuse between applications simplifying the development of large multi-disciplinary projects. By breaking the link between the problem definition and execution we are able to find the optimal implementation on the currently available computational resources.

Having built an application from components with integral performance models we are able to make sophisticated scheduling decisions and match an application to the appropriate computational resources. This will become essential for the emerging heterogeneous computational grids that will dominate high performance computing in the future.

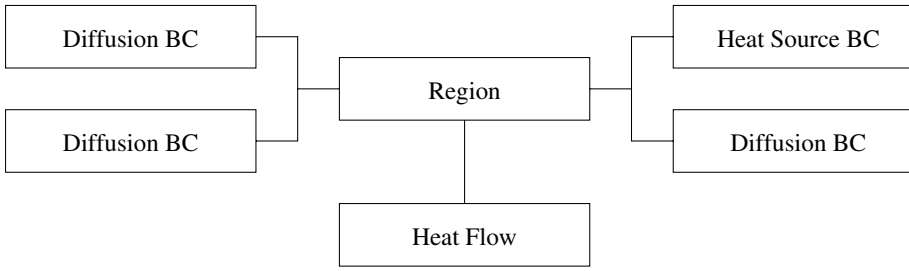


Fig. 3. The Component Network extracted from the graphical Problem Solving Environment.

Satin: Efficient Parallel Divide-and-Conquer in Java

Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal

Dept. of Mathematics and Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands
rob@cs.vu.nl, kielmann@cs.vu.nl, bal@cs.vu.nl
http://www.cs.vu.nl/albatross/

Abstract. Satin is a system for running divide and conquer programs on distributed memory systems (and ultimately on wide-area metacomputing systems). Satin extends Java with three simple Cilk-like primitives for divide and conquer programming. The Satin compiler and runtime system cooperate to implement these primitives efficiently on a distributed system, using work stealing to distribute the jobs. Satin optimizes the overhead of local jobs using on-demand serialization, which avoids copying and serialization of parameters for jobs that are not stolen. This optimization is implemented using explicit invocation records. We have implemented Satin by extending the Manta compiler. We discuss the performance of ten applications on a Myrinet-based cluster.

1 Introduction

There is currently much interest in divide and conquer systems for parallel programming [2, 6, 10, 11, 15]. Divide and conquer style programs start by dividing the problem into subproblems. Each subproblem is then recursively solved, again by dividing it into smaller subproblems. An example of such a system is Cilk [6], which extends C with divide and conquer primitives. Cilk runs these annotated C programs in parallel, in an efficient way, but is mainly targeted at shared memory machines. Atlas [2], an extension of Java, is a divide and conquer system designed for distributed memory machines. Its primitives have a high overhead, however, so it runs fine-grained parallel programs inefficiently.

In this paper, we introduce a new system, called Satin, which also is a divide and conquer system based on Java. Satin (as the name suggests) was inspired by Cilk. In Satin, single-threaded Java programs are parallelized by annotating methods that can run in parallel. Our ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area clusters (e.g., the DAS [8]). We think that the divide and conquer model will map efficiently on such systems, as the model is also hierarchical. In this paper, however, we focus on the implementation of Satin on a single local cluster computer. In contrast to Atlas, Satin is designed as a compiler-based system in order to achieve high performance. Satin is based on the Manta [12] native compiler, which supports highly efficient serialization and communication. Parallelism is achieved in

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 690–699, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Satin by running different spawned method invocations on different machines. The system load is balanced by *work stealing*.

One of the contributions we make in the paper is the use of explicit invocation records, to enable the on-demand serialization of parameters to spawned method invocations. This optimization is possible because of Satin's parameter semantics. Furthermore, we demonstrate that Satin can run efficiently on distributed memory machines. Satin also cleanly integrates divide and conquer programming into Java, and solves some problems that are introduced by this integration (e.g., by garbage collection).

2 The Programming Model

Satin's programming model is an extension of the single-threaded Java model. Satin programmers thus need not use Java's multithreading and synchronization constructs or Java's Remote Method Invocation mechanism, but can use the much simpler divide and conquer primitives described below.

2.1 Spawn and Sync

We have introduced three new keywords to the Java language, `spawn`, `sync`, and `satin`. The `spawn` keyword must be placed in front of a method invocation, which will then be called a *spawned method invocation*. When `spawn` is placed in front of a method invocation, conceptually a new thread is started which will run the method. (The implementation of Satin, however, eliminates thread creation altogether.) The spawned method will run concurrently with the method that executed the `spawn`. In Satin, spawned methods always run to completion. The `sync` operation waits until all spawned calls in this method invocation are finished. The return values of spawned method invocations are undefined until a `sync` is reached. The `satin` modifier must be placed in front of a method declaration, if this method is ever to be spawned.

To illustrate the use of `spawn` and `sync`, an example program is shown in Fig. 1. This code fragment calculates Fibonacci numbers, and is a typical example of a divide and conquer program. Note that this is a benchmark, and *not* a suitable algorithm for efficiently calculating the Fibonacci numbers. The program is parallelized just by inserting `spawn` in front of the recursive calls to *fib*. The two subproblems will now be solved concurrently. Before the results are combined, the method must wait until both subproblems have actually been solved, and have returned their value. This is done by the `sync` operation. A well known optimization in parallel divide and conquer programs is to make use of a threshold on the number of spawns. When this threshold is reached, work is executed sequentially. This approach can easily be programmed using Satin.

Satin does not provide shared memory, because this is hard to implement efficiently on distributed memory machines. Moreover, our ultimate goal is to run Satin on wide-area systems, which clearly do not have shared memory. The only way of communicating between threads is via the parameters and the return

```
class Fibonacci {
    SATIN int fib(int n) {
        if(n < 2) return n;

        int x = SPAWN fib(n - 1);
        int y = SPAWN fib(n - 2);
        SYNC;

        return x + y;
    }

    public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        int result = f.fib(10);
        System.out.println("Fib 10 = " + result);
    }
}
```

Fig. 1. A Satin example: Fibonacci.

value. The parameter passing mechanism, as described in Sect. 2.2, assures that all data that can be accessed via parameters will be sent to the machine that executes the spawned method invocation.

2.2 The Parameter Passing Mechanism

Because Satin does not provide shared memory, objects passed as parameters in a spawned call to a remote machine will not be available on that machine. Therefore, Satin uses *call-by-value* semantics when the runtime system decides that the method will be spawned remotely. This is semantically similar to the standard Java Remote Method Invocation (RMI) mechanism [17]. Call-by-value is implemented using Java's *serialization* mechanism, which provides a *deep copy* of the serialized objects [16]. For instance, when the first node of a linked list is passed as an argument to a spawned method invocation (or a RMI), the entire list is copied.

It is important to minimize the overhead for work that does not get stolen and is executed by the machine that spawned the work, as this is the common case. For example, in almost all applications we have studied so far, at most 1 out of 400 jobs gets stolen. Because copying all parameter objects (i.e., using call-by-value) in the local case would be prohibitively expensive, parameters are passed by reference when the method invocation is local. Therefore, the programmer cannot assume either call-by-value or call-by-reference semantics for `satin` methods (normal methods are unaffected and have the standard Java semantics). It is therefore erroneous to write Satin methods that depend on the parameter passing mechanism. (A similar approach is taken in Ada for parameters of a structured type.)

An important characteristic of Satin is that when the extensions `satin`, `spawn`, and `sync` are removed, a sequential standard Java program remains.

This program produces the same result as the parallel Satin program. This always holds, because Satin does not specify the parameter passing mechanism. Using call-by-reference in all cases (as normal Java does) is thus correct.

3 The Implementation

The large majority of jobs will not be stolen, but will just run on the machine the jobs were spawned on. Therefore, it is important to reduce the overhead that the Satin runtime system generates for such jobs as much as possible. The key problem here is that the decision whether to copy the parameters must be made at the moment the work is executed or stolen, not when the work is generated. To be able to defer this important decision, Satin's runtime system uses invocation records, which will be described below. The large overhead for creating threads or building task descriptors (copying parameters) was also recognized in the lazy task creation work by Mohr et al. [13].

When a program executes a `spawn`, Satin redirects the method call to a stub. This stub creates an *invocation record* (see Fig. 2), describing the method to be invoked, the parameters that are passed to the method, and a reference to where the method's return value has to be stored. For primitive types, the value of the parameter is copied. For reference types (objects, arrays, interfaces), only a reference is stored in the record. In the example of Fig. 2, a `satin` method is invoked with an integer, an array, and an object as parameters. The integer is stored directly in the invocation record, but for the array and the object, references are stored, to avoid copying these data structures. The compiler allocates space for a counter on the stack of all methods executing `spawn` operations. This counter is called the *spawn counter*, and counts the number of pending spawns, which have to be finished before this method can return. The *address* of the spawn counter is also stored in the invocation record.

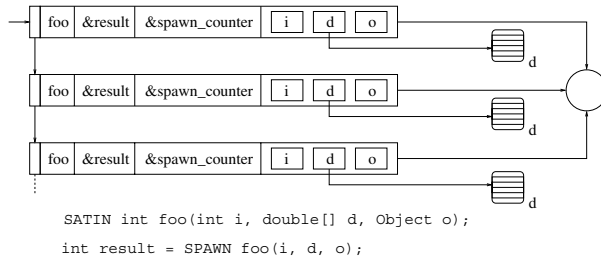


Fig. 2. Invocation records in the job queue.

The stub that builds an invocation record for a spawned method invocation is generated by the Manta compiler, and is therefore very efficient, as no runtime type inspection is required. From an invocation record, the original call can

be executed by pushing the value of the parameters (which were stored in the record) onto the stack, and by calling the Java method.

The invocation record for a `spawn` operation is stored in a queue. The spawn counter (located on the stack of the invoking method) is incremented by one, indicating that the invoking method now has a pending spawned method invocation. The invoking method may then continue running. After the spawned method invocation has eventually been executed, its return value will be stored at the return address specified in the invocation record. Next, the spawn counter (the address of which is also stored in the invocation record) will be decremented by one, indicating that there now is one less pending `spawn`. The `sync` operation executes work stored in the job queue, and waits for the spawn counter to become zero. When this happens, there are no more pending spawned method invocations, so the method may continue.

Serialization is Java's mechanism to convert objects into a stream of bytes. This mechanism always makes a deep copy of the serialized objects: all references in the serialized object are traversed, and the objects they point to are also serialized. The serialization mechanism is used in Satin for marshaling the parameters to a spawned method invocation. Satin implements serialization on demand: the parameters are serialized only when the work is actually stolen. In the local case, no serialization is used, which is of critical importance for the overall performance. In the Manta system, the compiler generates highly-efficient serialization code. For each class in the system a so-called serializer is generated, which writes the data fields of an object of this class to a stream. When an object has reference fields, the serializers for the referenced objects will also be called. Furthermore, Manta uses an optimized protocol to represent the serialized objects in the byte stream. Manta's implementation of the serialization mechanism is described in more detail in [12].

The invocation records describing the spawned method invocations are stored in a double ended job queue. A Dijkstra-like protocol [6] is used to avoid locking in the local case. Satin registers the invocation records at the garbage collector, keeping parameter objects alive when they are referenced only via the invocation record, and not via a Java reference. (Otherwise, the garbage collector might free objects that are needed to execute the spawn operations, but are no longer referenced via the Java program). Satin's work stealing is implemented on top of the Panda communication library [1], primarily using Panda's message passing primitives. On the Myrinet network (which we use for our measurements), Panda is implemented on top of the LFC [3] network interface protocol. Satin uses the efficient, user level locks that Panda provides for protecting the work queue.

4 Performance Evaluation

We evaluated Satin's performance using ten application kernels. All measurements were performed on a cluster of the Distributed ASCI Supercomputer (DAS), each containing 200 MHz Pentium Pros that are locally connected by Myrinet. The machines run the Linux (RedHat 6.2) operating system.

4.1 Basic Spawn Overhead (Fibonacci)

An important indication of the performance of a divide and conquer system is the overhead of the parallel application on one machine, compared to the sequential version of the same application. The sequential version is obtained by filtering the keywords `satin`, `spawn`, and `sync` out of the parallel program. The difference in run times between the sequential and parallel programs is caused by the creation, the en-queuing and de-queuing of the invocation record, and the construction of the stack frame to call the Java method. Fibonacci gives an indication of the worst-case overhead, because it is very fine grained. Cilk is very efficient, the parallel Fibonacci program on one machine has an overhead of only a factor of 3.6 (measured on a Sun Enterprise 5000, with 167 MHz UltraSPARC processors) [6]. Atlas is implemented completely in Java and does not use on-demand serialization. Therefore its overhead is much worse, a factor of 61.5 (hardware unknown) [2]. The overhead of Satin is a factor 7.25, substantially lower than that of Atlas.

These overhead factors can be reduced at the application level by introducing threshold values that spawn only large jobs. For Fibonacci, for example, we tried a threshold value of 20 for a problem of size 45, so all calls to *fib(n)* with $n<20$ are executed sequentially, without using `spawn`. This simple change to the application reduced the overhead to almost zero. Still, $22.8 \cdot 10^6$ jobs were spawned, leaving enough parallelism for running the program on large numbers of machines. For Fibonacci, the threshold can easily be determined by the programmer, while for other applications this may be difficult or impossible. In general, however, it still is important to keep the sequential overhead of a divide and conquer system as small as possible, as it allows the creation of more fine-grained jobs and thus a better load balancing.

The overhead for the other applications we implemented is much lower than for the (original) Fibonacci program, as shown in Table 1. Here, t_s denotes the run time of the sequential program, t_1 the run time of the parallel program on one machine. In general, the overhead depends on the number of parameters to spawned methods. All parameters have to be stored in the invocation record when the work is spawned, and pushed on the stack again, when executed.

4.2 Parallel Applications

We ran ten applications on the DAS cluster, using up to 32 CPUs. Figure 3 shows the achieved speedups while Table 2 provides detailed information about the parallel runs. All speedup values were computed relative to the sequential applications, with the Satin-specific annotations removed from the code.

There is a strong correlation between measured speedup and the sequential overhead value, as already shown in Table 1: the lower the overhead, the higher the speedup we achieved. In Table 2 we compare the measured speedup with its upper bound, computed as the number of CPUs divided by the overhead on a single CPU. We also show the percentage of this upper bound as actually achieved by the measured speedup. This percentage is very high for most applications,

Table 1. Application overhead factors, times in seconds.

application	problem size	t_s	t_1	overhead
adaptive integration	0, 2.0E5, 1.0E-4	363.137	451.117	1.24
set covering problem	58, 29	1983.723	2071.333	1.04
fibonacci	41	65.517	475.133	7.25
fibonacci threshold	45	473.749	473.834	1.00
Iterative deepening A*	60	220.131	250.001	1.14
knapsack problem	28	1064.220	1150.016	1.08
matrix multiplication	1024 x 1024	137.982	141.742	1.03
n over k	34, 17	971.991	977.847	1.01
n-queens	15	1861.318	1909.942	1.03
prime factorization	1234567890	874.504	930.954	1.06
traveling sales person	17	982.864	1352.617	1.38

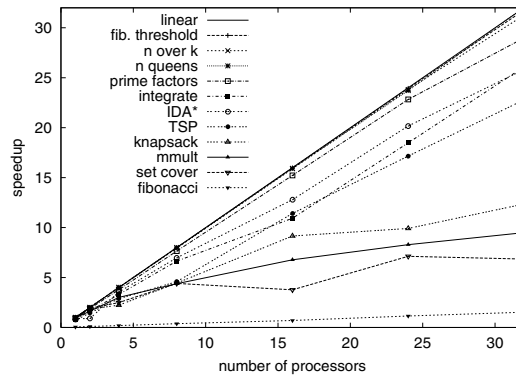


Fig. 3. Application speedups

Table 2. Parallel performance breakdown for 32 CPUs

application	overhead	speedup	#CPUs/ overhead	% max speedup	jobs	stolen
integrate	1.24	26.09	25.81	101 %	$63.3 \cdot 10^6$	2187
set cover	1.04	6.85	30.8	22.2 %	$51.0 \cdot 10^6$	579
fibonacci	7.25	4.31	4.4	98.0 %	$536 \cdot 10^6$	2906
fib. threshold	1.00	31.77	32.0	99.3 %	$22.8 \cdot 10^6$	1951
IDA*	1.14	25.82	28.1	91.9 %	$33.6 \cdot 10^6$	3866
knapsack	1.08	12.36	29.6	41.8 %	$33.5 \cdot 10^6$	417
mmult	1.03	9.49	31.1	30.5 %	$37.4 \cdot 10^3$	8567
n over k	1.01	31.27	31.7	98.6 %	$1.05 \cdot 10^6$	2458
n-queens	1.03	31.02	31.1	99.7 %	$2.47 \cdot 10^6$	3027
prime factors	1.06	28.98	30.2	96.0 %	$33.6 \cdot 10^6$	2609
tsp	1.38	22.79	23.2	98.2 %	$200 \cdot 10^6$	3026

denoting that Satin's communication costs are low. The actual percentage depends (like the sequential overhead) on the number of method parameters and their total serialized size. Table 2 also lists the total number of spawned jobs and the number of stolen jobs, which is less than 1 out of 400 for all applications, except for mmult. Because the number of stolen jobs is so small, speedups are mainly determined by sequential overhead. A good example is Fibonacci, which achieves 98% of the upper bound, but still has a low speedup due to the sequential overhead. Satin's sequential efficiency thus is important for the successful deployment of the divide and conquer paradigm for parallel computing.

Mmult does not get good speedups, because the problem size is small due to memory constraints, the run time on 32 cpus is only 14 seconds. Also, much data is transferred, in total over all CPUs, 31 MByte is sent per second. The mediocre speedup of knapsack, a very irregular application, is caused by load imbalance. The search space is pruned by both the weights and the values of the elements in the knapsack, making it difficult to estimate the grain size of a job. Therefore, many small jobs get stolen. The same holds for the set-covering problem, where a large percentage of the time is spent in finding work. On 32 nodes, only 1.2 percent of the work stealing attempts were successful.

5 Related Work

We discussed Satin, a divide and conquer extension of Java. Satin has been designed for distributed memory machines, while most divide and conquer systems use shared memory machines (e.g. Cilk [6]). There is also a version of Cilk for distributed memory machines, called CilkNOW [5], but it only supports functional Cilk programs (without shared memory), and it does not make a deep copy of the parameters to spawned methods. Our own previous work on parallel divide and conquer [9] was based on the C language while having similar restrictions as CilkNOW. Alice [7] and Flagship [18] offer a hardware solution for parallel divide and conquer programs (e.g., a reduction machine with one global address space for the parallel evaluation of declarative languages), while Satin is purely software based, and does not require, or provide, a single address space.

Mohr et al. [13] describe the importance of avoiding thread creation in the common, local case (lazy task creation). Satin also avoids creating threads in the local case, targeting distributed memory adds the problem of copying the parameters of parallel invocations (marshalling). Satin builds on the ideas of lazy task creation, and avoids both the starting of threads and the copying of parameter data by choosing a suitable parameter passing mechanism.

Another divide and conquer system based on Java is Atlas [2]. Atlas is not a Java extension, but a set of Java classes that can be used to write divide and conquer programs. While Satin is targeted at efficiency, Atlas was designed with heterogeneity and fault tolerance in mind, and aims only at a reasonable performance. Because Satin is compiler based, it is possible to generate code to create the invocation records, thus avoiding all runtime type inspection. The

Java classes presented in [11] can also be used for divide and conquer algorithms. However, they are restricted to shared-memory systems.

A compiler-based approach is also taken by Javar [4]. In this system, the programmer uses annotations to indicate divide and conquer and other forms of parallelism. The compiler then generates multi-threaded Java code, which runs on any JVM. Therefore, Javar programs run only on shared memory machines and DSM systems, whereas Satin programs run on distributed memory systems. Java threads impose a large overhead, which is why Satin does not use threads at all, but provides light weight invocation records. There are many other projects which use Java for parallel processing, for instance [14] and the work referenced in this paper.

6 Conclusions and Future Work

We have described our experiences in building a parallel divide and conquer system for Java, which runs on distributed memory machines. We have shown that an efficient implementation is possible by choosing convenient parameter semantics. An important optimization is the on-demand serialization of parameters to spawned method invocations. This was implemented using explicit invocation records. Our Java compiler generates code to create these invocation records for each spawned method invocation. We have also demonstrated that divide and conquer programming can be cleanly integrated into Java, and that problems introduced by this integration (e.g., through garbage collection) can be solved.

Our ultimate goal is to use Satin for distributed supercomputing applications on hierarchical wide-area clusters. We believe that divide and conquer programs will map efficiently on such systems, as the model is also hierarchical. Our intention is to carry out research on the scheduling of divide and conquer programs on hierarchical wide-area systems.

Acknowledgments

This work is supported in part by a USF grant from the Vrije Universiteit. The wide-area DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Aske Plaat for his contribution to this research, and Ronald Veldema, Jason Maassen, Ceriel Jacobs, and Rutger Hofman for their work on the Manta system. We thank Kees Verstoep and John Romein for keeping the DAS in good shape. We also thank the anonymous referees for their useful comments on this paper.

References

[1] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[2] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[3] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[4] A. Bik, J. Villacis, and D. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, November 1997.

[5] R. Blumofe and P. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *In Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, Anaheim, California, 1997.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 207–216, Santa Barbara, California, July 1995.

[7] J. Darlington. Alice: a multi-processor reduction machine for the parallel evaluation of applicative languages. In Arvind, editor, *1st Conference on Functional Programming Languages and Computer Architecture*, pages 65–76, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.

[8] The Distributed ASCI Supercomputer (DAS). http://www.cs.vu.nl/das/.

[9] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[10] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *SuperComputing '99*, November 1999.

[11] D. Lea. A java fork/join framework. In *ACM Java Grande 2000 Conference*, San Francisco, California, June 2000.

[12] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[13] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[14] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, Nov. 1997.

[15] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 72–83, Atlanta, May 4-6 1999. Massachusetts Institute of Technology.

[16] Sun MicroSystems, Inc. Java (TM) Object Serialization Specification, 1996. ftp://ftp.javasoft.com/docs/jdk1.1/serial-spec.ps.

[17] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July–September 1998.

[18] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. In *15th IEEE/ACM Symp. on Computer Architecture*, pages 124–130, Honolulu, Hawaii, 1988. ACM SIGARCH newsletter,16(2).

Implementing Declarative Concurrency in Java

Rafael Ramirez, Andrew E. Santosa, and Lee Wei Hong

National University of Singapore
School of Computing
S16, 3 Science Drive 2, Singapore 117543
Tel. +65 8742909, Fax +65 7794580
{rafael,andrews,leeweiho}@comp.nus.edu.sg

Abstract. We describe the implementation of a high-level language based on first order logic for expressing synchronization in multi-threaded Java programs. The language allows the programmer to *declaratively* state the system safety properties as temporal constraints on specific program points of interest (*events*). The constraints are enforced by the runtime environment, i.e. the program points of interest are traversed only in the order specified by the constraints. The implementation is based on the incremental and *lazy* generation of partial orders among events. Although the implementation reported in this paper is concerned only with the synchronization of Java programs, the general underlying synchronization model we present is *language independent* in that it allows the programmer to glue together separate concurrent threads regardless of their implementation language and application code.

1 Introduction

The task of programming concurrent systems is substantially more difficult than the task of programming sequential systems in respect of both correctness (to achieve correct synchronization) and efficiency. One of the reasons is that it is very difficult to separate the concurrency issues in a program from the rest of the code. Synchronization concerns cannot be neatly encapsulated into a single unit which results in their implementation being scattered throughout the source code. This harms the readability of programs and severely complicates the development and maintenance of concurrent systems. Furthermore, the fact that program synchronization concerns are intertwined with the rest of the code, also complicates the formal treatment of the concurrency issues of the program which directly affects the possibility of formal verification, synthesis and transformation of concurrent programs. In Java, for instance, a common problem of writing multi-threaded applications is that synchronization code ensuring data integrity tends to dominate the source code completely. This produces code which is difficult to understand, modify and treat formally [11].

We believe that the system concurrency issues are best treated as orthogonal to the system base functionality. In this paper we describe the implementation of a first order language in which the safety properties of concurrent Java programs

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 700–708, 2000.
© Springer-Verlag Berlin Heidelberg 2000

are declaratively stated as constraints. Basic Java programs are annotated at points of interest so that the run-time environment enforces specific temporal constraints between the visit times of these points. The declarative nature of the constraints provides great advantages in writing, reasoning and deriving concurrent programs [4,5]. The constraints are *language independent* in that the application program can be specified in any conventional programming language. The model has a procedural interpretation which is based on the *incremental* and *lazy* generation of constraints, i.e. constraints are considered only when needed to reason about the execution order of current events.

Section 2 describes some related work. Section 3 presents the language used for specifying the synchronization constraints of concurrent Java programs. In Section 4 we describe the implementation of the language and finally Section 5 summarizes the contributions and indicates some areas of future research.

2 Related Work

Various attempts have been done by the object-oriented community to separate concurrency issues from functionality. Recently, some researchers have proposed *aspect-oriented programming* (*AOP*) [6]. In this area, the closest related work is the work by De Volder and D'Hondt [14]. Their proposal utilizes a full-fledged logic programming language as the aspect language. In order to specify concurrency issues in the aspect language, basic synchronization declarations are provided which increase program readability. Unfortunately, the declarations have no formal foundation. This reduces considerably the declarativeness of the approach since correctness of the program concurrency issues directly depend on the implementation of the the declarations.

Closer to our work are *path expressions* (e.g., PROCOL [2]) and constructs similar to *synchronization counters* (e.g., Guide [8] and DRAGOON [1]). These proposals, as ours, differ from the AOP approach in that the specification of the system concurrent issues are part of the final program. Unfortunately, synchronization counters have limited expressiveness since it is not possible to order methods explicitly. Path expressions are more expressive in this respect but they cannot express some important synchronization (e.g., producers-consumers) without embedding guards that increases complexity.

An important issue is that in all of the other proposals mentioned above, method execution is the smallest unit of concurrency. This is impractical in actual concurrent programming where we often need finer-grained concurrency.

On the other hand, several approaches to incorporate declarative programming into concurrent programming have been proposed. Traditional approaches to declarative concurrent programming include concurrent logic programming (e.g. Parlog [3], KL1 [13]) and concurrent constraint programming [12]. Although these approaches preserve many of the benefits of the abstract declarative model, such as the logical reading of programs and the use of logical terms to represent data structures, important program properties, namely safety and progress properties, remain implicit. These properties have to be preserved by using con-

trol features such as modes and sequencing, producing programs with little or no declarative reading. Also, in such languages, there is no clear separation of program application functionality and concurrency control.

3 Logic Programs for Concurrent Programming

3.1 Events and Constraints

Many researchers, e.g. [7,9], have proposed methods for reasoning about temporal phenomena using partially ordered sets of events. Our approach to concurrent programming is based on the same general idea. The basic idea here is to use a constraint logic program to represent the (usually infinite) set of constraints of interest. The constraints themselves are of the form $X < Y$, read as "X precedes Y" or "the execution time of X is less than the execution time of Y", where X and Y are events, and $<$ is a partial order.

The constraint logic program is defined as follows[1]. Constants range over events classes $E, F, \ldots$ and there is a distinguished (postfixed) functor $+$. Thus the terms of interest, apart from variables, are $e, e+, e++, \ldots, f, f+, f++, \ldots$. The idea is that e represents the first event in the class E, $e+$ the next event, etc. Thus, for any event X, $X+$ is implicitly preceded by X, i.e. $X < X+$. We denote by $e(+N)$ the N-th event in the class E. Programs facts or *predicate constraints* are of the form $p(t_1, \ldots, t_n)$ where p is a user defined predicate symbol and the t_i are ground terms. Program rules or *predicate definitions* are of the form $p(X_1, \ldots, X_n) \leftarrow B$ where the X_i are distinct variables and B a rule body whose variables are in $\{X_1, \ldots, X_n\}$. A program is a finite collection of rules and is used to define a family of partial orders over events. Intuitively, this family is obtained by unfolding the rules with facts indefinitely, and collecting the (ground) *precedence constraints* of the form $e < f$. Multiple rules for a given predicate symbol give rise to different partial orders. For example, since the following program has only one rule for p:

$p(e, f).$
$p(E, F) \leftarrow E < F,\ p(E+, F+).$

it defines just one partial order $e < f$, $e+ < f+$, $e++ < f++, \ldots$. In contrast,

$p(e, f).$
$p(E, F) \leftarrow E < F,\ p(E+, F+).$
$p(E, F) \leftarrow F < E,\ p(E+, F+).$

defines a family of partial orders over $\{e, f, e+, f+, e++, f++, e+++\ldots\}$. We will abbreviate the set of clauses $H \leftarrow Cs_1$, ..., $H \leftarrow Cs_n$ by the *disjunction constraint* $H \leftarrow Cs_1; \ldots; Cs_n$ (disjunction is specified by the disjunction operator ';').

[1] For a complete description, see [10]

3.2 Markers and Events

In order to refer to the visit times at points of interest in the program we introduce markers. A marker declaration consists of an event name enclosed by angle brackets, e.g. `<e>`. Markers annotations can be seen simply as program comments (i.e. they can be ignored) if only the functional semantics of an application is considered. Markers are associated with programs points between instructions, possibly in different threads. Constraints may be specified between program points delineated by these markers. For a marker M, $time(M)$ (read as "the visit time at M") denotes the time at which the instruction immediately preceding M has just been completed. In the following, we will refer to $time(M)$ simply by M whenever confusion is unlikely. Given a pair of markers, constraints can be stated to specify their relative order of execution in all executions of the program. If the execution of a thread T_1 reaches a program point whose execution time is constrained to be greater than the execution time of a not yet executed program point in a different thread T_2, thread T_1 is forced to suspend execution. In the presence of loops and procedure calls a marker is typically visited several times during program execution. Thus, in general, a marker M associated with a program point p represents an event class E where each of its instances $e, e+, e++\ldots$ corresponds to a visit to p during program execution (e represents the first visit, $e+$ the second, etc.).

3.3 Example

An example discussed in almost every textbook on concurrent programming is the producer and consumer problem. The problem considers two types of processes: producers and consumers. Producers create data items (one at a time) which then must be appended to a buffer. Consumers remove items from the buffer (if it is not empty) and consume them, i.e. perform some computation which uses the data item. Thus, producer processes can be defined by an infinite cycle containing *produce* (producing an item) and *append* (appending the item to the buffer). Similarly, consumer processes can be defined by an infinite cycle containing *remove* (removing an item from the buffer) and *consume* (consuming the item). Producers and consumers may be defined as follows (the code has been annotated with markers $p1$, $p2$, $c1$ and $c2$).

```
class Producer extends Runnable
{ ...
  public void run() {
    while (true) {
      produce(X);
      < p1 >
      append(X);
      < p2 >
    } }
... }
```

```
class Consumer extends Runnable
{ ...
  public void run() {
    while (true) {
      < c1 >
      remove(X);
      < c2 >
      consume(X);
    } }
... }
```

If we assume an infinite buffer, the only safety property needed is that the consumer never attempts to remove an item from an empty buffer. This property can be expressed by

$p(p2, c1).$
$p(P, C) \leftarrow P < C,\ p(P+, C+).$

In practice however, buffers are finite. Thus, in practice, a producer is allowed to append items to the buffer only when the buffer is not full. For instance, this safety property for a system with a buffer of size 3 can be expressed by

$p(c2, p1 + ++).$
$p(C, P) \leftarrow C < P,\ p(C+, P+).$

4 Implementation

A prototype implementation of the ideas presented here has been written using the language Java. Java was used both to implement the constraint language and to write the code of a number of applications. We discuss the implementation in this section.

4.1 Architecture

The architecture of our implementation (Figure 1) consists of four main parts:

- The *interface* is an object of class *Tempo* which decides whether or not threads suspend upon reaching a marker during execution. When a thread reaches a marker m, a request is sent to the interface to determine whether the current event e associated with m is *disabled*, i.e. it appears on the right of a precedence constraint $X < e$, or *enabled*, i.e. otherwise, wrt the *constraint store*. If e is found to be disabled, the thread is blocked until e becomes enabled, otherwise the thread proceeds execution at the instruction immediately after m.
- The *constraint store* contains the system synchronization constraints. As shown in Figure 1, it can be decomposed into two parts: the predicate definition store (DS) and the constraint store (CS). The constraint store contains precedence, predicate and disjunction constraints.
- The *user program* is the main program and typicly specifies the system synchronization constraints, creates the Tempo object and spawn a number of threads which may contain markers.
- The *verifier* examines the specification of the system synchronization constraints to detect any errors such as infinite loops in predicate definitions, e.g. $p(X) \leftarrow p(X)$, before the Tempo object is created.

The overall mechanism is as follows: Once the Tempo object has been created and a set of threads have been spawned, whenever one of the threads, say T,

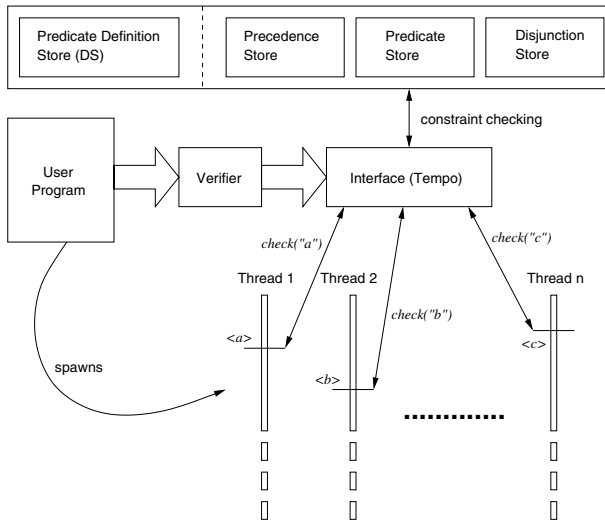


Fig. 1. Implementation architecture

reaches a marker in their code, a communication between the thread and the constraint store is triggered. Currently the communication is implemented as a request from T to the Tempo object. The request is of the form $check(Str)$, in which Str is a string denoting marker's identifier (e.g., "$p1$," "$p2$," "$c1$," "$c2$" in the producers-consumers example).

4.2 The Constraint Interpreter

The decision on whether to suspend a thread or not is based on a procedural interpretation of the constraint logic programs used to specify synchronization constraints. The procedural interpretation allows a correct specification to be executed in the sense that events are only executed as permitted by the constraints represented by the program. This procedural interpretation is based on an incremental execution of the program and a *lazy* generation of the corresponding partial orders. Constraints are generated by the constraint logic program only when needed to reason about the execution times of current events. Figure 2 shows the complete interpreter for a constraint logic program specifying the synchronization constraints of a Java program. The algorithm is implemented in the class *Tempo*.

To see how the interpreter works, let us use the producers-consumers example explained in Section 3.3. Initially, DS contains the predicate definition $p(X,Y) \leftarrow X < Y, p(X+,Y+)$, and CS contains two predicate constraints: $p(p2,c1)$ and $p(c2,p1++)$. Suppose a consumer thread reaches marker $c1$. At this point, the constraint store is checked to determine whether $c1$ is enabled or disabled which requires the expansion of $p(p2,c1)$. Based on the predicate definition in DS, $p(p2,c1)$ is expanded to $p2 < c1, p(p2+,c1+)$ at which point

```
place system constraints in CS;
place predicate definitions in DS;
when thread T reaches marker E:
    get current occurrence e of
    the event class associated with E;
    while e is neither enabled nor
    disabled in CS (e appears in
    user-defined constraint C):
        replace C in CS by its definition;
        if e is disabled in CS
        then suspend T
        if e is enabled in CS
        then execute e
        else if e is conditionally enabled in CS
            then reduce CS by e;
                execute e
            else if e is unknown in CS
                then error.
execute e:
    for each precedence constraint in CS with
    e on the left e < R (including those inside
    disjunctions):
        delete e < R;
        resume threads waiting for R.

reduce CS by e:
    for each disjunction D in CS:
        delete all alternatives of D in which e
        is disabled.

for a disjunction D:
if e is enabled in
every alternative of D
    then e is enabled in D
else if e is disabled in
every alternative of D
    then e is disabled in D
else if e is enabled in
some alternatives of D
and disabled in all others
    then e is conditionally
    enabled in D
else e is unknown in D.

for a conjunction CS:
if e is disabled in at least
one constraint in CS
    then e is disabled in CS
else if e is unknown in at
least one constraint in CS
    then e is unknown in CS
else if e is conditionally
enabled in at least one
constraint in CS
    then e is conditionally
    enabled in CS
else e is enabled in CS.
```

Fig. 2. Synchronization constraints interpreter

$c1$ is found to be disabled since it appears on the right hand side of the precedence constraint $p2 < c1$. Thus, execution of the consumer thread is suspended at marker $c1$. At some point, a producer thread reaches marker $p1$ and after checking the constraint store it is determined that $p1$ is enabled (it does not appear on the right hand side of any precedence constraint). It then proceeds to add an item to the buffer, and then it reaches marker $p2$. Since $p2$ is enabled execution of the producer thread proceeds, constraint $p2 < c1$ is deleted from CS since it has already been satisfied, and threads suspended for event $c1$ are awaken so they can re-checks the constraint store. At this point, since $c1$ does not appear on the right side of any precedence constraints, the consumer thread may continue its execution. This has the effect of not allowing retrieval of items when the buffer is empty. A similar situation occurs when a producer thread attempts to add an item to a full buffer.

Our implementation is still in a prototype stage, thus several efficiency issues have still to be addressed. The main issue is the time taken for a thread to check whether a marker is enabled or not. Ideally, this time should be close to zero. In our current implementation, the check is linear to the size of the constraint store which produces an acceptable performance of the system. We are investigating further improvements to the implementation including porting it to a more efficient programming language.

Fairness is implicitly guaranteed by our implementation. Every event that becomes enabled will eventually be executed (provided that the program point associated with it is reached). This is implemented by dealing with event execution requests in a first-in-first-out basis. Although fairness is provided as the default, users, however, may intervene by specifying priority constraints among events. It is therefore possible to specify unfair scheduling.

5 Conclusion

We have described the implementation of a high-level language based on first order logic for expressing synchronization constraints in multi-threaded Java programs. In the language, the safety properties of the system are *explicitly* stated as temporal constraints. Programs are annotated at points of interest so that the run-time environment enforces specific temporal relationships between the visit times of these points. Constraints are *language independent* in that the application program can be specified in any conventional concurrent object-oriented language. The constraints have a procedural interpretation that allows the specification to be executed. The procedural interpretation is based on the incremental and *lazy* generation of constraints, i.e. constraints are considered only when needed to reason about the execution time of current events.

This paper presents work in progress so several important issues are still to be considered. Our implementation is still in a prototype stage, thus several efficiency issues have still to be addressed. In particular, we will focus on how the two key features of incrementality and laziness may be most efficiently achieved. Another important issue is how to deal with progress properties. Currently, constraints explicitly state all safety properties of programs. However, the progress (liveness) properties of programs remain implicit. It would be desirable to be able to express these properties explicitly as additional constraints, but so far we have not devised a way to do that. Future versions may also include deadlock detection feature. We are considering a mechanism that checks user constraints for cycles (e.g., $A < B, B < A$) whenever a timeout occurred. We are also looking into developing a methodology that uses the generative technique for engineering multi-threaded Java programs using constraints. Using this technique, programs may be generated using high-level descriptions. The declarative nature of our language particularly fits this approach.

References

1. Atkinson, C. 1991. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach.* Addison-Wesley.
2. Van den Bos, J. and Laffra, C. 1989. *PROCOL: A parallel object language with protocols.* ACM SIGPLAN Notices 24(10):95–112, October 1989. Proc. of OOPSLA '89.
3. Gregory, S. 1987. *Parallel Logic Programming in PARLOG*, Addison-Wesley.
4. Gregory, S. and Ramirez, R. 1995. *Tempo: a declarative concurrent programming language.* Proc. of the ICLP (Tokyo, June), MIT Press, 1995.

5. Gregory, S. 1995. *Derivation of concurrent algorithms in Tempo*. In LOPSTR95: Fifth International Workshop on Logic Program Synthesis and Transformation.
6. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. 1997. Aspect-oriented programming. In *ECOOP '97—Object-Oriented Programming*, Lecture Notes in Computer Science, number 1241, pp. 220–242, Springer-Verlag.
7. Kowalski, R.A. and Sergot, M.J. 1986. A logic-based calculus of events. New Generation Computing 4, pp. 67–95.
8. Krakowiak, S., Meysembourg, M., Nguyen Van, H., Riveill, M., Roisin, C. and Rousset de Pina, X. 1990. *Design and implementation of an object-oriented strongly typed language for distributed applications.* Journal of Object-Oriented Programming 3(3):11–22.
9. Pratt, V. 1986. Modeling concurrency with partial orders. International Journal of Parallel Programming 15, 1, pp. 33–71.
10. Ramirez, R. 1996. *A logic-based concurrent object-oriented programming language*, PhD thesis, Bristol University.
11. Ramirez, R., Santosa, A. E. 2000. *Declarative concurrency in Java*, Proceedings of the 5th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2000). Springer-Verlag.
12. Saraswat, V., and Rinard, M. 1990. *Concurrent Constraint Programming.* Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages. pp.232-245.
13. Ueda, K. and Chikayama, T. 1990. *Design of the kernel language for the parallel inference machine.* Computer Journal 33, 6, pp.494-500.
14. De Volder, K. and D'Hondt, T. 1999. Aspect-oriented logic meta programming. In *Meta-Level Architectures and Reflection*, Lecture Notes in Computer Science number 1616, pp. 250–272. Springer-Verlag.

Building Distributed Applications Using Multiple, Heterogeneous Environments*

Paul A. Gray[1] and Vaidy S. Sunderam[2]

[1] University of Northern Iowa, Dept. of Mathematics,
Cedar Falls, IA, 50614-0506, USA
gray@cns.uni.edu

[2] Emory University, Dept. of Mathematics and Comp. Sci.,
Atlanta, GA, 30322, USA
vss@mathcs.emory.edu

Abstract. There continues to be genuine interest in expanding application environments to include geographically-distributed resources and to be able to dynamically re-configure the environment to suit the needs of the application. The expansion of an application's resource pool and dynamicity of the environment introduces fresh nuances in programming paradigms and demands novel characteristics of the substrate exportable to the application. This paper describes our research efforts toward the goal of providing applications with such an environment; one that is able to actively adopt and relinquish resource pools, is able to pre-configure and dynamically re-configure attributes to suit the needs of the application, and maintains reliability and persistence in the presence of failures. This paper discusses our findings relating to portability and utility aspects of *shared-library* usage over heterogeneous environments.

1 Introduction

Distributed computing environments have evolved considerably in the last decade.

The *Parallel Virtual Machine* ([5] environment and the *Message-Passing-Interface* ([13]) (MPI) are two noteworthy projects that have evolved over this time frame. PVM and MPI provide users with an abstract virtual machine (VM) and tools to manage the collection of resources underneath the abstraction. With the APIs that these packages provide, application programmers are given a well-defined and robust environment for distributed and parallel application development. Indeed, the virtual machine approach has been shown to be a very compelling paradigm for a wide variety of applications. However, there are some limitations. For example, one limitation is that the current virtual machine approach limits a process' interaction with entities outside of the virtual machine. Processes running in two distinct virtual environments are unable to use the virtual machine paradigm to interact. Another related limitation is the lack of a

* Research supported in part by NSF grant ACI-9872167

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 709–717, 2000.
© Springer-Verlag Berlin Heidelberg 2000

model whereby processes inside of a virtual machine may interact with "open" services such as database and web servers, ftp services and other services which are so prevalent in the networking aspects of today. Further, the issue of *process migration* is one that is also fundamentally difficult in such environments, where process-naming conventions and check-pointing issues are paramount issues ([12]). Simply stated, naming difficulties and resource-specific attributes such as open files and sockets present good arguments for binding processes to a single resource.

In general, the VM paradigm possesses many strong features and will continue being a steadfast aspect of distributed computing in the future. At the same time, the commodity networks and workstations upon which the VM paradigm has its stronghold is evolving at such a rapid rate that commensurate evolutions in the fundamentals of virtual machines are necessary. Microprocessors which drive todays workstations keep getting faster – a lot faster. Further, the cost of a moderately-configured workstation in today's market is quite reasonable, which allows one easy access to off-the-shelf, state-of-the-art componentry. This is most evident in the growing number of SMP workstations, the wide-reaching grasp of computational grids, and the proliferation of "Beowulf"-type cluster environments available in the market today. All of this means that the future of distributed computing will need to support traditional applications as well as servers, SMPs, and various configurations of clusters. They will also need to be able to adjust to underlying attributes so as to allow application-specific optimizations; being able, for example, to detect and leverage upon a Non-Uniform Memory Access (NUMA) configuration, the availability of high-speed communication attributes such as Myrinet ([1]), and so forth.

It is our view that a strong trend in distributed computing exists that incorporates a deep awareness of the underlying architectures of its component resources and also includes the utilization of multiple virtual environments, which are brought together for collaborative or enterprise-type utilization. In abstract terms, virtual environments *merge* together for a specific collaborative phase and are then able to *split* apart, in-tact, along arbitrary boundaries. Once merged, applications that were previously restricted to one environment are able to collaborate and to utilize the resources found on the complement environment (left). Further, if permitted, applications are able to instantiate *new* processes on any capable resource in the combined environment. Figure 1 also shows that these resources may split apart arbitrarily (right). The succession of groups of resources and applications is determined by applications, the owners of the resources or even by faults in the network environment. The processes in the succeeding environments are left in-tact. These processes continue with their designated processing duties, provided they are able and permitted to do so. Note that if the fissure of the environment was caused by a failure in the network, the separate groups may re-merge when the network recovers. Thus, this would allow for more resilient processes, which would be able to survive and recover from environmental catastrophes. Note also that the component environments may re-merge at a later time or merge with alternate environments at any time.

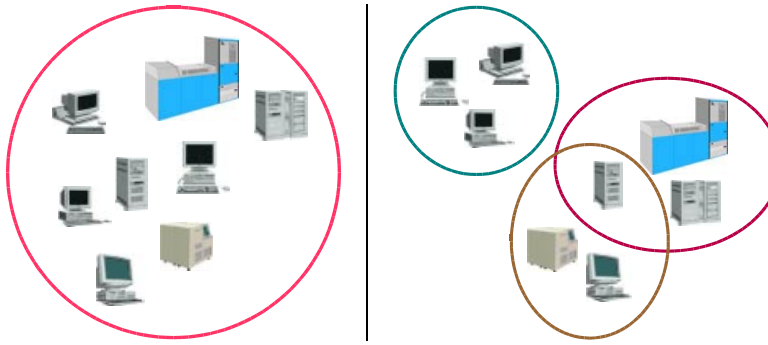


Fig. 1. *The merging of virtual environments provides applications access to the resources and applications on the complement. Virtual environments are also permitted to split apart along arbitrary boundaries as illustrated in the depiction on the right.*

In creating such an environment, methodologies are needed that will enable distinct parallel applications running on top of the environment to discover each other, to synchronize and permit communication, to promote collaboration, and to cleanly detach from each other upon separation.

The subject of this paper focuses on the specific task of creating processes upon the resources of the complementary environment once merging of the resources has occured. The difficulties which arise involve foreign architectures which cause binary incompatibilities and locating the appropriate binary executables appropriate for the remote architectures. The approach presented here involves details of how environments are explored in order to detect compatibilities and missing attributes and how the environment may be dynamically re-configured so as to include all of the facilities that a process may need prior to its instantiation. Our approach utilizes both exploration using Java-based front ends and dependence on shared libraries formats for the soft-installation of processes, described in detail in the next section.

2 Designing Dynamic Environments

As mentioned above, one of the issues that comes up when discussing the merging of two distinct resources pools is the utilization of these newly-acquired resources. Take for example, the issue of instantiating a new process; that is the task of physically loading a process upon a remote resources. Resource pools are not assumed to share any common filesystem, nor are they assumed to be binary compatible. Referring back to Figure 1, suppose that the resources within the environment on the left were of different architectures; some running Windows NT as their operating system, others consisting of Sun Sparcs and DEC Alphas running Solaris and Linux as their operating systems. These entities would be unable to instantiate native forms of processes on the complement environment

without some help. This process of probing the application's needs, exploring the attributes of the environment, re-configuring the environment to make it able to support the application, and the ultimate instantiation of a process onto a remote system is referred to as the *soft-installation* of a process.

The soft-installation process that we've employed involves two main phases: a Java-based setup and exploration phase; and phase that involves the locating, relocating, and loading of shared libraries needed by the process. These two phases are explained below. A major prototype for the soft-installation mechanism is part of our project termed "IceT" ([10], [11], [9]), and the sections below describe the manner in which code portability is achieved through the use of Java and shared libraries. The investigations to date involve applications which consist of a Java-based application wrapper and C, C++ or FORTRAN computational substrates. The two are linked together using the Java Native Interface (JNI) and encapsulation of the C, C++, or FORTRAN codes into shared libraries.

2.1 The Role of Java

The main attribute of the Java programming language which we make use of is the bytecode representation. In bytecode form, a Java process is significantly platform independent. We leverage upon this attribute of Java in the process of "handshaking," i.e. as a way as a way to "introduce" and declare the needs of the application to a remote resource. The major portion of the application code is wrapped by a Java class. The Java-based application wrapper is glued to the application using the Java Native Interface (JNI) and Java-based classes which make calls into the native code. The underlying application code is compiled in the form of a shared library and linked into the Java application wrapper using the Java directive: "`System.loadLibrary("foo")`," where "*foo*" is the name of the application core in its shared library form.

The simple Java call:

```
static{ System.loadLibrary("foo")}
```

is embedded in the Java application wrapper's compiled bytecode. It becomes part of the application's static bytecode representation which is passed between resources and used for handshaking. That is, the need for the shared library "foo" is embedded in the bytecode that we use for the handshaking process[1]. By dis-assembling the bytecode instruction, one can elicit the name of the shared library required for this process' execution. This describes the methodologies used for the soft-installation process. The application's bytecode wrapper is dis-assembled and analyzed for shared library components that will be needed for the process to run. For more details on the detection of the library calls and on the dis-assembly process involving the Java bytecode, see [7].

Once the shared library requirements are determined, the next step is to locate the appropriate form of the shared library for the resource and, if necessary,

[1] The embedding of the shared library call into the Java bytecode is part of the Java Virtual Machine definition, and accomplished by the standard Java compiler. No special pre- or post-processing of the bytecode is necessary.

to soft-install the shared library locally. This scenario is depicted in Figure 2. The use of the "foo" shared library is detected by the parsing of the Java bytecode front end, described above. The next task is to fill this foo library requirement with the appropriate binary form, based upon the operating system and architecture of the host system.

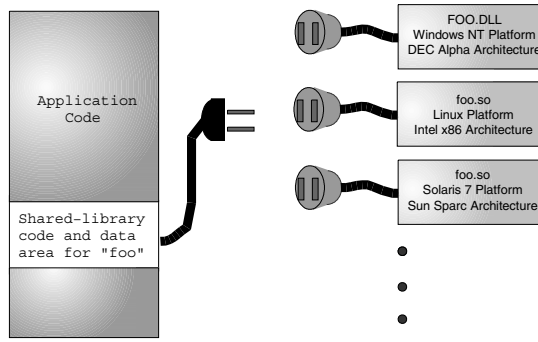


Fig. 2. *Since shared libraries are not embedded in the static form of the application's executable, one is able to locate, transport, and "plug-in" these library dependencies according to the particular environment upon which the application is to run.*

Once a shared library requirement "foo" is detected from the bytecode analysis, the environment looks first to determine if the shared library foo is already present; "FOO.DLL" in the case of a Windows environment or "libfoo.so" in the case of a Unix environment. In the event that the foo library is not found locally, a query is made to a local library server as to the location of the "foo" library for the appropriate operating system (say Windows NT or Linux) and platform (say a DEC Alpha or Intel x86). If the appropriate form of the shared library is unavailable, the creation of the process fails.

2.2 Shared Library Aspects

It may seem like many of the attributes of the environment that we seek can be delivered by the Java runtime environment itself: Heterogeneous execution, portability, and such are all associated with the Java programming language. However, there are many reasons why a pure Java implementation falls short. A Java program simply cannot access the low-level aspects of a system which are ultimately necessary for high-performance.

While the environment that we seek to provide allows for highly portable codes, application performance is of equal concern. The environment should facilitate high-performance applications, written to take specific advantage of specialized software. That is, the environment under development will be able to provide portability and high performance gains to applications that depend upon

high-performance tools such as Nexus ([4]), the Virtual Interface Architecture (VIA) ([3]), numerical applications based upon LINPACK routines ([2]) and so on.

To illustrate the feasibility of such an implementation, experiments that we have performed using this paradigm of Java-wrapped shared libraries include the facilitation of applications based upon highly-tuned BLAS (Basic Linear Algebra Subroutine), LINPACK routines, CCTL (a high-performance multicast communication package) ([7]) and LAM-MPI ([6]). The performance seen in these native-language packages remains unattainable in Java. It should also be mentioned that there is a slight performance penalty for using Java as an application wrapper as well, and determining the extent of the penalty is also part of our investigations ([8]).

An ultimate goal of the environment under development is the transparent utilization of resources in a combined, heterogeneous environment. A major fissure that we have had to cross is when the environment contains both Microsoft Windows and Unix-based workstations (Solaris and/or Linux, for example). This presents a major roadblock inasmuch as the inner workings of the respective operating systems are vastly different. However, both operating systems share enough commonalities so as to permit the mutual soft-installation of a single application built upon shared libraries onto Windows and Unix platforms using the technique outlined above.

2.3 Shared and Static Libraries

Generally speaking, executable programs are applications which receive and process messages, signals, and user input. Typically, libraries are not directly executable and do not receive messages or signals directly. Libraries are separate files which contain functions that can be called by applications and other libraries so as to perform certain tasks. In languages such as C, C++, and FORTRAN, application code is linked with various libraries. The result is an executable file.

These libraries which application code are linked may be *static* or *shared*. In the Windows environment, shared libraries take the form of *dynamic link libraries* or ".DLL"s, but the fundamental attributes of dynamic link libraries and shared libraries in Unix environments are the same. Shared libraries in Unix environments are typically identified by a "`.so`" extension. When an application is linked to a static library, the code within that library becomes an inseparable part of the application's executable file. Changes made to a static library after the linking process would not be reflected in the executable form of the application. When an application is linked to a static library, the library function calls are resolved at link time. In contrast, when an application utilizes a shared library, linking is done at run time. Shared libraries do not become intertwined with the executable form of the application in the way a static library is embedded into the an application's executable form. This allows changes made to the library core to be incorporated into the application without re-compilation or changes made of the application. The operating system loads the shared library into memory and resolves the calls into the library during the execution of the application.

These are fundamental differences between shared and static libraries. Another major difference occurs when the application is running. A major difference is that static library codes linked into an application are loaded into memory as part of the executing application whereas shared library code is able to be loaded into memory on-demand. This feature of shared libraries facilitates our mechanism for soft-installation as we can link the appropriate library with our applications *on* the host where the application is to be executed and *at* the time the application requires it.

Other differences between static and shared libraries effect the manner in which one writes an application. Each application that is linked to a static library embeds a copy of the static library's codes into its own executable. Thus, every application has its own copy of the static library code necessary for its execution. A shared library, on the other hand, exists uniquely. Applications that have linked to a shared library simultaneously share the single instance of the library. That is, if several applications are running at the same time, the shared library code is loaded only once into the environment, and mutually shared and accessible to each application that it has been linked to. Thus, care must be taken when one writes a shared library so as to avoid unintended manipulation of the shared library's resources. So, while building a shared library may be as simple as changing a compiler flag, there are some significant attributes of shared libraries that effect how applications are executed.

The singular shared library aspect disadvantage is evident in the intrinsic way an application interacts and links with a shared library. If two or more applications that utilize the same shared library are running simultaneously, the shared library is loaded into the operating system's kernel only once. Thus, each application is sharing a single copy of common code and resources. For this reason, it is necessary that the code in the shared library be *reentrant*. As there is a single shared library instance shared by all applications depending upon it, global variables in the shared library code are at risk of being manipulated simultaneously by independent processes — a situation to be avoided at all costs. This puts additional semaphore and mutex coding responsibilities on the application programmer.

3 Conclusion

There are significant benefits to shared-library utilization in the soft-installation approach. Shared libraries are linked independently of the applications that use them which allows us to detect and relocate shared libraries as necessary and permits the use of highly-tuned libraries that are maintained outside of the application. Our preliminary experimentation in providing portability based upon shared library usage has been very successful.

However, shared library dependence also brings with it some disadvantages which need to be considered when developing applications, such as the issues of *reentrant* and *thread safe* code. These disadvantages are not insurmountable. First, the primary disadvantage is that shared libraries are much more difficult

to develop: One must be aware of the aspects which allow a single instance of the library to be utilized by multiple applications at the same time. Thus, a necessary aspect of shared libraries which are able to be utilized by multiple applications is the that the library must be reentrant.

Preliminary investigations have shown feasibility and proof of concept for the portability and distribution of processes as described in this paper. These experimentations include extending portability to BLAS, CCTL, LINPACK, and MPI libraries. Our ongoing investigations involve incorporation of user authentication, process validation, and pinpointing the conflicts in a multiple-application, single-library setting.

References

1. Nanette Boden, Danny Cohen, Robert Felderman, Alan Kulawik, C. L. Seitz, and J. N. Seizovic. Myrinet: A gigabit per second local area network. *IEEE-Micro*, 15(1), February 1995.
2. Jack Dongarra, Jim Bunch, Cleve Moler, and Pete Stewart. The FORTRAN–based LINPACK routines. Available from NAG, Downers Grove, IL.
3. Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2), March/April 1998.
4. I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.
5. G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computing Conference*, pages 258–261. IEEE, 1991.
6. Vladimir Getov, Paul Gray, Sava Mintchev, and Vaidy Sunderam. Multi-language programming environments for high performance Java computing. *Scientific Programming*, 9(11):1161–1168, November 1999.
7. Vladimir Getov, Paul Gray, and Vaidy Sunderam. Aspects of portability and distributed execution for JNI-wrapped code. To appear in a special issue on MPI in Concurrency: Practice and Experience.
8. Vladimir Getov, Paul Gray, and Vaidy Sunderam. MPI and Java-MPI: Contrasts and comparisons of low-level communication performance. In *Proceedings of Supercomputing 99*, November 1999.
9. P. Gray and V. Sunderam. IceT: Distributed Computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1168, November 1997.
10. P. Gray and V. Sunderam. Native Language-Based Distributed Computing Across Network and Filesystem Boundaries. *Concurrency: Practice and Experience*, 10(1), 1998.
11. Paul Gray and Vaidy Sunderam. The IceT Environment for Parallel and Distributed Computing. In Y. Ishikawa, R. R. Oldehoeft, J. V. W. Reynders, and M. Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, number 1343 in Lecture Notes in Computer Science, pages 275–282, New York, December 1997. Springer Verlag.

12. M. Litzkow, M. Livny, and M. W. Mutka. Condor – A hunter of idle workstations. In *Oriceedubgs if the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.
13. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI, The Complete Reference*. MIT Press, November 1995.

A Multiprotocol Communication Support for the Global Address Space Programming Model on the IBM SP

Jarek Nieplocha Jialin Ju Tjerk P. Straatsma
Pacific Northwest National Laboratory, Richland, WA 99352, USA
http://www.emsl.pnl.gov/docs/parsoft/armci

Abstract. The paper describes an efficient communication support for the global address space programming model on the IBM SP, a commercial example of the SMP (symmetric multi-processor) clusters. Our approach integrates shared memory with active messages, threads and remote memory copy between nodes. The shared memory operations offer substantial performance improvement over LAPI, IBM one-sided communication library, within an SMP node. Based on the experiments with the SPLASH-2 LU benchmark and a molecular dynamics simulation, our multiprotocol support for the global address space is found to improve performance and scalability of applications. This approach could also be used in optimizing the MPI-2 one-sided communication on the SMP clusters.

1 Introduction

This work is motivated by applications that require support for a shared-memory programming style rather than just message passing. Many of them are characterized by irregular data structures, and dynamic or unpredictable data access patterns. For certain types of applications, the shared-memory programming model can be substituted or supported with the global address space model. Systems with a global address space usually do not offer coherent shared memory at the operating system level. Instead, in a distributed-memory environment they provide remote memory operations, for example as in the SHMEM library [1] on the Cray T3E, or one-sided communication operations in the MPI-2. The global address space can be used directly by applications, or indirectly supporting a shared-memory view of data structures emulated through a user-level library interface, such as the Global Arrays (GA) [2], that transparently to the user performs an appropriate translation of shared to distributed memory references. The programming model based on the global address space has the added benefit of preserving data locality information (the application is explicitly aware of, and controls data distribution) that well-written distributed-memory message-passing applications can exploit to increase performance. It also allows access the data in a fashion similar to shared memory without the interprocess synchronization imposed by the traditional cooperative message passing model.

In recent years, clustered systems with symmetric multi-processor (SMP) nodes have become increasingly popular as the cost effectiveness of SMP nodes and high-performance networks improved. An example of such an architecture is the IBM SP, a distributed-memory machine with SMP nodes, a network supporting high-performance user-space communication, and a rich programming environment that offers active messages and remote memory copy through the LAPI system library, threads, thread-safe MPI, and the standard Unix shared-memory interfaces within the

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 718-728, 2000.
© Springer-Verlag Berlin Heidelberg 2000

SMP nodes. Our goal is to optimize communication support for the global address space on clustered SMP-based systems for applications that use one process/task per processor (rather than the explicitly multithreaded single process with per SMP node).

The main contribution of this paper is an integration of multiple communication protocols such as active messages, threads, and remote memory copy with shared memory to support the global address space model efficiently. Performance of one-sided operations is substantially improved by mapping the global address space to the shared-memory segments on SMP nodes. This technique makes a low-level messaging library such as LAPI responsible only for internode communication while the intranode communication is handled directly by shared memory. Performance advantages of this approach are shown for the remote-memory operations in the ARMCI portable communication library[3] and two applications, the SPLASH-2 LU benchmark and NWChem molecular dynamics code. The performance of NWChem was improved by 12.6% on 64 processors by relinking this large and already well tuned application unchanged with a different version of the ARMCI library. Although this paper focuses on the IBM SP, we used the multiprotocols for supporting global address space through ARMCI on other cluster platforms. We chose the SP since it 1) is a major cluster platform widely used for technical computing, 2) offers a richer set of vendor supported protocols (including active messages) relevant to our objectives than most other cluster platforms, and 3) all the discussed protocols are widely available in the user mode and supported on the current SP system configurations.

Shared memory has been exploited before in low-level one-sided messaging systems like Active Messages[4] or Nexus [5], and higher-level two-sided message-passing interfaces like MPI [6] on SMP clusters. However, as the programming models based on the global-address space and message passing are fundamentally distinct, different strategies are needed to pass benefits of shared memory to the applications. For example, the IBM implementation of MPI on SMP nodes exploits shared memory to move data between a pair of MPI tasks through an internal buffer in shared memory with one task copying data into the shared memory buffer and then the other copying into its separate address space. ARMCI places the application data in shared memory thus the data can be accessed without any intermediate memory copies and message buffer management overheads. It helps achieve 15 times better latency and 67% better bandwidth than in the vendor version of MPI within the SMP node. Similar performance gains are realized through the shared memory in ARMCI over LAPI within the node. On the IBM SP, we found that the shared memory optimizations benefit more the remote memory/one-sided operations than the message passing.

The rest of this paper is organized as follows: Section 2 describes integration of shared memory with LAPI and thread-based protocols; Section 3 reports the results of basic communication operations; Section 4 presents experimental results of two applications, the SPLASH-2 LU benchmark and the molecular dynamic simulation; Section 5 discusses related work. Finally, we conclude in Section 6.

2 SMP-Aware Communication Protocols

We developed ARMCI[3] to support remote memory operations in the context of distributed array libraries such as GA and compiler run-time systems such as the Parallel Runtime Consortium [7] Adlib. ARMCI supports remote memory copy, accumulate, and synchronization operations. It is portable and compatible with message-passing libraries such as MPI or PVM. Unlike most existing similar facilities,

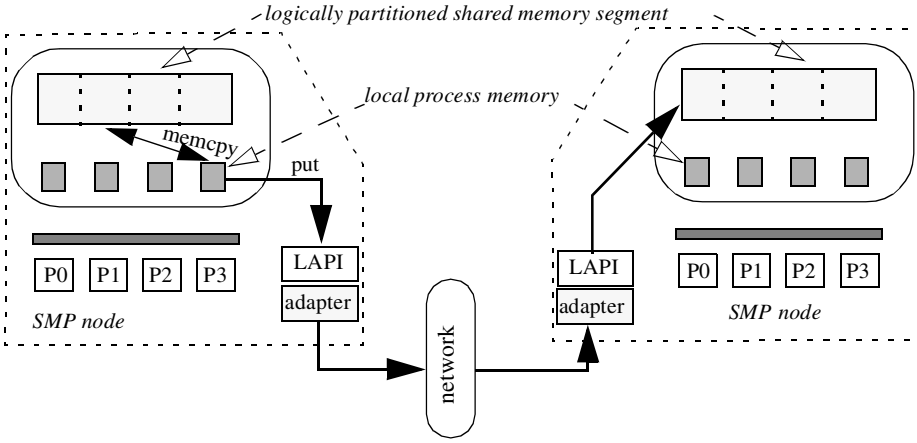


Figure 1: Combining shared memory with LAPI on the IBM SP

such as Cray SHMEM, it focuses on the noncontiguous data transfers that correspond to the data structures used in scientific applications (sections of multi-dimensional dense or sparse arrays, scatter, gather). Such transfers are optimized thanks to the non-contiguous data interfaces available for ARMCI data transfer operations: multi-strided and generalized UNIX I/O vector interfaces. ARMCI offers a simpler model and lower-level interface than the MPI-2 one-sided communication.

The standard parallel programming environment of the IBM SP includes LAPI, a low-level one-sided communication system that supports active messages (AM) and remote memory copy operations. LAPI offers competitive performance to MPI; however, unlike MPI it does not support noncontiguous data transfers which are common in scientific codes. ARMCI on the IBM SP uses active messages and threads rather than remote memory copies to optimize noncontiguous data transfers. Each process in a LAPI program uses at least three threads: the main application thread, and two extra threads for executing the LAPI active message handlers. In applications that use MPI in addition to LAPI, there are three additional threads introduced by the thread-safe IBM MPI library. For example, on the 4-way SMP node with 4 user processes (explicitly single threaded) that use MPI and LAPI, there are 4x6=24 threads. LAPI is supported in the SMP environment. However, since it uses the network adapter [8] to move data between processes on the same SMP node as if they were on separate nodes, its performance is not optimal. In particular, to transfer data between address spaces of two processes, LAPI performs at least two memory copies (to and from the adapter DMA area), and in many cases generates an interrupt.

We developed hybrid communication protocols for the ARMCI to exploit the SMP locality information and shared memory communication. The locality information is determined at start time from the process-to-node mapping, and used to select appropriate communication protocols. Within the SMP node, all operations are implemented on top of shared memory rather than with LAPI, see Figure 1. For inter-node communication, depending on the request size and shape, ARMCI chooses between active messages (AM) or remote memory copies to optimize bandwidth, see Figure 2. An implementation of the gather operation used by ARMCI is described in [9]. Since ARMCI includes a memory allocation routine to be used in the context of remote memory copy, memory is allocated using the System V shared memory

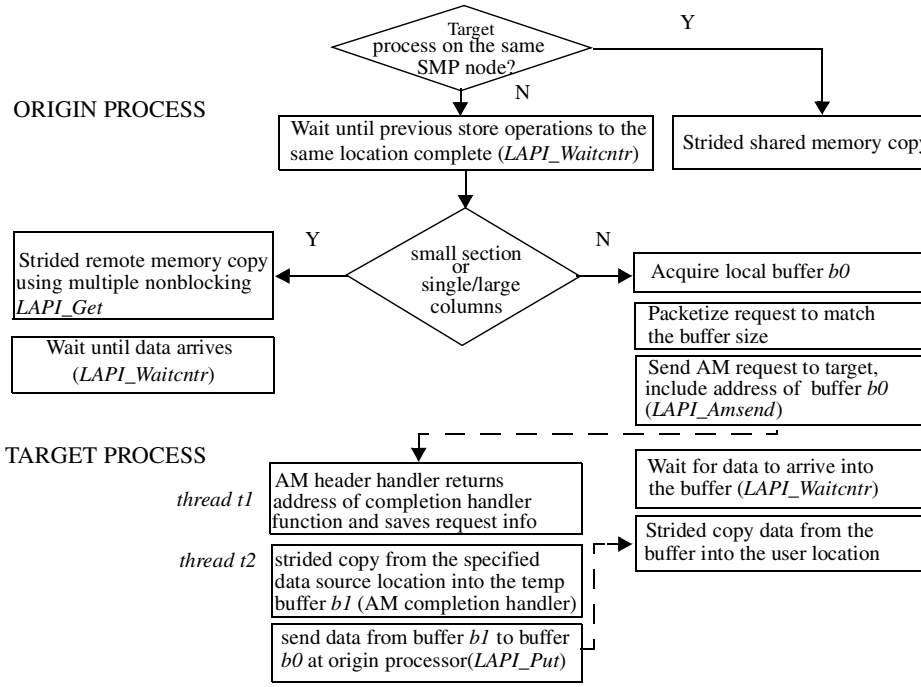


Figure 2: SMP-aware implementation of the ARMCI strided get operation on the IBM SP

interfaces. This critical function helps reduce the *put*/*get* operations to a memory copy and avoid at least one memory copy that LAPI must do. As explained below, the main difficulty is posed by the requirement to make multithreaded active-message internode protocols compatible with intranode shared memory operations.

ARMCI, in addition to remote memory copy, supports atomic operations: accumulate (reduction) as well as read-modify-write. The mutual exclusion embedded in the semantics of these operations requires special care when hybrid protocols are used. ARMCI on uniprocessor nodes uses active messages, threads, and Pthread mutexes. Since the same memory region in address space of process A can be addressed concurrently by a process B executing on the same SMP node and process C on a remote node, the mutual exclusion primitives must synchronize multiple threads in both the same and different process spaces. In AIX, Pthread mutexes cannot be used in this context. We designed a mutex lock replacement for both thread mutexes and System V semaphores. It offers a very low overhead and is free of the disadvantages of spin locks that can waste substantial amounts of CPU time by not yielding the processor when the mutex cannot be acquired for an extensive amount of time. We use the AIX atomic operation *check_lock* to check the content of a word. If the mutex is already acquired by another thread we use a spin lock with limited asymptotic backoff before finally yielding the processor to another thread.

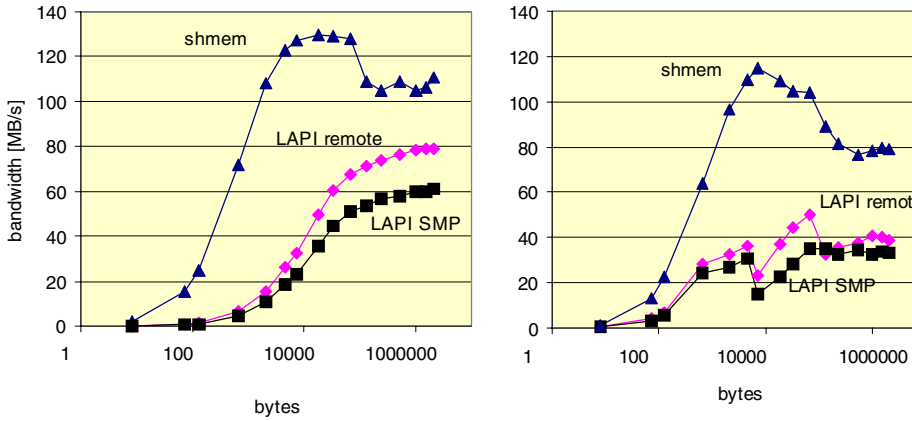


Figure 3: Performance of contiguous get (left) and contiguous accumulate (right) operations implemented using shared memory(shmem) and LAPI on the same SMP or remote node.

3 Performance of Communication Operations

We used a 16-node, 4-way SMP IBM SP with 64 PPC-604e processors and the TB3MX adapter at PNNL to study the performance of remote memory operations, the SPLASH-2 LU kernel benchmark and the molecular dynamics simulation. In this section, we discuss performance of the ARMCI get and accumulate operations in accessing contiguous and strided data on the same and remote SMP node. For the same node, the performance using the LAPI-based protocols and the shared-memory operations is presented.

Figures 3-4 demonstrate that our SMP-aware protocol outperforms LAPI by a large margin within a node. Interestingly, the observed bandwidth in the LAPI protocols used for intranode communication is in many cases worse than for the internode communication. This phenomenon is attributed to the fact that LAPI uses the network adapter even when communicating within the same node. For the intranode communication, the same adapter is used for sending and receiving the data, whereas for the internode communication the adapter handles only one side of the data transfer.

Despite the obvious differences between one-sided protocols in ARMCI and two-sided point-to-point message-passing protocols in MPI, we present performance of these systems to demonstrate how these interfaces take advantage of shared memory. Unlike LAPI, the IBM implementation of MPI already uses shared memory for communication within the SMP node. Table 1 shows latency and bandwidth numbers for the ARMCI get and the MPI send/receive operations. In this test, we used contiguous data transfers and tried to assure that the data is not already in cache. The MPI latency below is reported as 1/2 of the roundtrip time between two tasks for 1-byte message size, and we measured bandwidth for 512KB data size. The primary reason ARMCI outperforms MPI by a wide factor within the SMP node is that ARMCI get operation reduces simply to a memory copy whereas the MPI protocol adds the cost of message queue management and requires two memory copies and cooperation of two tasks to move data through an MPI internal shared memory buffer. Of course,

these advantages apply to the intranode environment. For the internode communication ARMCI uses LAPI, and in this case MPI and LAPI performances are similar [10].

The SMP-aware protocols, by improving communication performance within an SMP node, expose one additional layer of memory hierarchy in the IBM SP and provide an additional optimization opportunity to the applications.

Table 1: Performance of MPI send/receive and ARMCI get on the SMP node

interface	latency [µs]	bandwidth [MB/s]
MPI	13.2	65.63
ARMCI	0.85	109.64

4 Application Study

We used the SPLASH-2 LU benchmark and the NWChem molecular dynamics code to study the implication of SMP-aware communication protocols on the application performance. Neither of these codes was developed to exploit performance characteristics of the SMP-based clustered systems, and in this respect, are representative of the majority of scientific parallel codes. Since both codes perform internode and intranode communication, it was not clear what degree of performance improvement should be expected.

4.1 LU SPLASH-2 Benchmark

The SPLASH-2 benchmark suite [11] is a set of parallel applications for use in the design and evaluation of shared-memory multiprocessing systems. The suite contains two types of codes: full applications and kernels. We chose the LU program, which is one of the kernel programs from SPLASH-2, to evaluate the performance of our approach. The LU program factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The factorization uses blocking to exploit temporal locality w.r.t. individual submatrix elements [12]. Originally designed to run on shared memory systems, this benchmark can only be used on a single SMP node of the IBM SP. Some modifications were needed to use the global address space model.

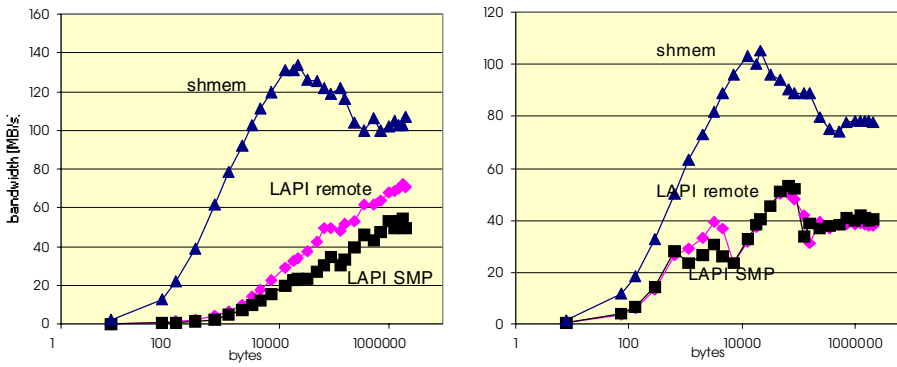


Figure 4: Performance of strided get (left) and strided accumulate (right) operations implemented using shared memory (shmem) and LAPI on the same SMP or remote node.

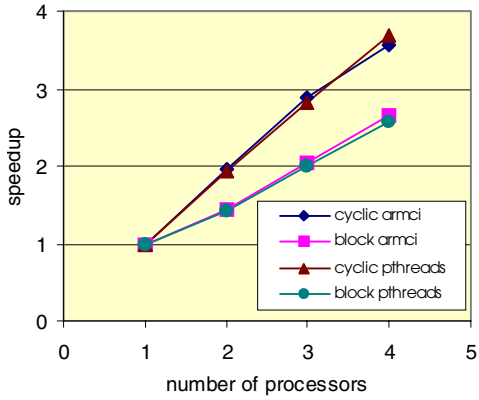


Figure 5: Speedup in the Pthread and ARMCI versions of the SPLASH-2 LU benchmark using block and block cyclic distributions on one SMP node

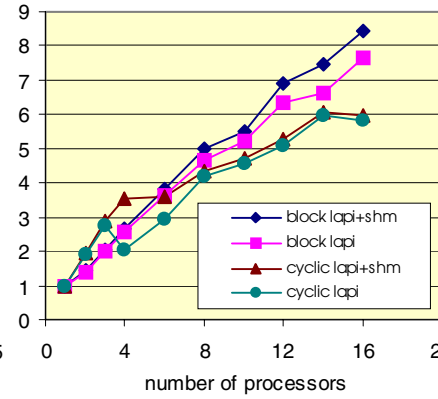


Figure 6: Speedup in the SPLASH-2 LU benchmark using block and block cyclic distributions and SMP-aware or SMP-oblivious communication protocols

We also developed a Pthread version of the benchmark to evaluate the performance of our modifications within an SMP node. The primary issues to be addressed included memory allocation, access to shared data, and interprocess/thread synchronization.

The Pthread version uses threads while the ARMCI version uses processes. In the first case, shared data is located in the process memory and accessed directly by the threads as needed. In order to replace shared memory with global address space, we had to divide the shared data, assign it to individual processes, and allocate the corresponding storage on each process. To synchronize processes, we used MPI_Barrier. Threads are synchronized with a Pthread mutex and a condition variable.

During the LU factorization, if required data blocks are in the local process memory, they are accessed directly. Otherwise, ARMCI_Get is used to copy the data block from the remote process that owns it to the local temporary storage. The computation requires transferring data blocks from the same row and column (for diagonal blocks). The original benchmark uses block cyclic distribution for load balancing. We also used block decomposition, as it had better locality of the data accesses. To take advantage of the SMP performance, the blocks are distributed according to a block pattern, such that the block that needs to be transferred has a better chance of residing in the local memory or neighboring memory on the same node. We used a matrix size of 3072 and a block size of 32 to study performance of the SPLASH-2 LU benchmark. The performance results given in Figure 5 indicate that the global address space and shared memory versions of the benchmark have similar performance for the same distribution types. The block cyclic distribution gives better performance than the block distribution because of better load balancing properties.

As the Pthread version works only on a single SMP node of the SP, we used only the ARMCI version of the benchmark on multiple nodes to study performance of the SMP-aware communication. We relinked the program with two alternative versions of ARMCI - one that uses only LAPI and the other that uses LAPI for internode communication and shared memory on a node. Since the block cyclic decomposition yields better load balancing the code scales better on a single SMP node. However, the lower memory reference locality compared to block decomposition causes

communication patterns to be spread out across all the nodes. Consequently, the benchmark performance on multiple nodes is better with block decomposition. To exploit performance advantages of both decomposition schemes, we also developed a third version of the benchmark that uses block cyclic decomposition on a single SMP node and block decomposition on multiple SMP nodes. Its performance is a good as its component protocols in their optimal operational regimes, see Figure 6.

The impact of using faster communication within SMP nodes is more significant with the block rather than block cyclic decomposition because it better exploits data locality. Improved performance is observed for the approach using LAPI and shared memory, although the difference is small when few processors are used. This is because more time is spent in computation than in communication. With more processors, the multi-protocol approach shows better performance, proving the benefits of shared memory support for the global address space.

Figure 6 shows some performance degradation effects in the LAPI-only version with four processors. We contribute this phenomenon to the hardware and software interaction involving the heavily used (for all the intranode communications) network adapter and scheduling of the 24 threads for a particular communication pattern in the benchmark. As our approach relieves the adapter from handling intranode communication any similar performance degradations have never been experienced.

4.2 Molecular Dynamics Simulation

NWChem is a massively parallel computational chemistry software package developed on top of Global Arrays (GA). This large (>600,000 code lines) package contains multiple methods for quantum-mechanical and classical computational chemistry including molecular dynamics. The recent version of GA employs ARMCI as its run-time system. GA maintains only the distributed array infrastructure, performs array index translation to the global address space, and implements collective array operations. All the GA one-sided communication is supported through ARMCI.

Molecular dynamics (MD) simulations typically evaluate atomic interactions within a specified cutoff distance. The implementation of the method in NWChem uses this locality of atomic interactions in the way the data is distributed on a distributed-memory MPP. This domain decomposition of the physical simulation volume exploits the locality of the atomic interactions such that all-to-all interprocessor communication is avoided. On computers where the cost of communication between processor pairs is not homogeneous, such as SMP clusters, an additional optimization of the domain decomposition is in principle possible for example by avoiding the assignment of physically distant parts of the simulation volume to processors capable of fast communication. In practice, this means that the locality in the simulation space is reflected in the choice of processor assignment. The processors on a given SMP node should typically be assigned adjacent parts of the simulation volume, for which communication requirements are always high.

Molecular dynamics simulations of liquid water were carried out with three versions of the code, see Figure 7. The application uses numbers of processors that are powers of 2. The original version is based on the ARMCI library that uses only LAPI for communication between and within SMP nodes. The other versions use the SMP-aware ARMCI library, with and without an additional permutation of the process numbers (inside GA) to improve communication locality based on our analysis of the MD communication patterns. No modification of the application code was needed.

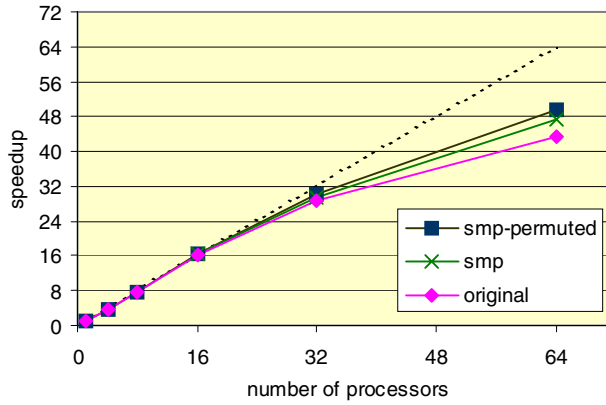


Figure 7: Performance of the MD simulation with SMP-aware and unaware (original) versions of ARMCI. A permutation of process numbers is used to improve communication locality

The three versions were produced by simply relinking the application with different versions of the ARMCI and GA libraries.

Similarly to the LU benchmark, the shared memory multiprotocols improved performance and scaling of this application. The improvement rate depends on the number of processors used and ratio of communication to calculation. The SMP optimizations are more effective when communication is made through a permutation of the process numbers. On 64 processors a 12.6% performance improvement is achieved over the original version. Since this well optimized application had already scaled almost linearly up to 32 CPUs, this is a substantial improvement, and it was achieved without any explicit modifications to this complex code. Moreover, 3-dimensional MD simulations on 4-way SMP clusters could not take full advantage of the communication locality. Our analysis indicates that with the increased number of CPUs per SMP node, the performance improvements should be even better.

5 Related Work

Multiprotocols involving shared memory has been used before to optimize performance of two- [6] and one-sided messaging systems [4,5]. Husbands and Hoe [6] used the shared memory mechanism for intra-SMP communication in an MPI implementation for a cluster interconnected by the StarT-X network. They optimized contiguous data transfer through a shared memory transfer facility in the MPICH channel layer on the node. ARMCI unlike MPI-1 supports one-sided communication. Lumetta *el.*, [4] proposed multi-protocol Active Messages on SMP cluster. Thir multi-protocol implementation of Active Message directs message traffic through the appropriate medium, either shared memory or network. Separate message queues are maintained for these two media. The shared memory queue block must first be mapped into address space of processes on the node. Message polling operations are ubiquitous in Active Message layers even with shared memory implementation, whereas ARMCI does not require polling. Also unlike [4], ARMCI incorporates threads among other protocols used. Foster, *el.*, [5] discussed the multi-protocol communication in the Nexus multi-threaded one-sided communication system that extends to heterogeneous platforms. Two-level protocols are available: one better suited for small, latency-

sensitive communications, and the other for large communications. The application performance can be optimized by using one link for synchronization and the other for data transfer. In ARMCI, selecting the protocol is not an issue anymore. Nexus unlike ARMCI does not include active messages among the protocols considered in [5].

The other differences between these papers and our work arise from the fact that ARMCI: 1) supports one-sided remote memory operations rather than one- or two-sided message-passing interfaces; 2) emphasizes noncontiguous data transfers (not addressed in papers [4-6]); and 3) supports atomic remote memory operations. In our experience, one-sided messaging libraries such as Nexus or Active Messages are very well suited for implementing inter-node communication but they add unnecessary overhead on the SMP node (e.g., related to message queue management, flow control, buffering) that can be avoided in the context of global address space model by using shared memory directly.

6 Conclusions and Future Work

We described a multiprotocol communication support for the global address space on the IBM SP that integrates shared memory within SMP nodes with the LAPI active messages, threads and remote memory copy between nodes. Shared memory offers substantial performance improvements over LAPI within a node for both contiguous and noncontiguous data transfers. Based on the SPLASH-2 LU benchmark and molecular dynamics simulation, the multiprotocol support for global address space is found to improve both performance and scalability of these applications. In the application context we have also found that 1) distribution plays an important role in exploiting the shared memory effectively and 2) replacing a shared memory programmming style (Pthreads) with the global address space model does not lead to performance losses within the SMP domain and provides good scaling across the cluster. Another important benefit of this approach is that the intranode communication is no longer involving the network adapter and its resources thus they can now be devoted exclusively to inter-node communications. This technique can also be used in implementations of the MPI-2 (not yet available on the IBM SP) on SMP clusters since it also offers a memory allocation interface (MPI_Alloc_mem) in the context of the one-sided operations. The described integrated protocols are employed in ARMCI for supporting the global address space model on more SMP-based systems than just the IBM SP. We intend to extend the ARMCI capabilities (e.g., support for heterogenous systems) and use it to implement other interfaces. For example, ARMCI is well suited for a portable implementation of the SHMEM library and we are also considering using it to support the MPI-2 1-sided “passive communication model”.

References

1. R. Barriuso, Allan Knies, SHMEM User’s Guide, Cray Research, SN-2516, 1994.
2. J. Nieplocha, R. J. Harrison, R.J. Littlefield. Global Arrays: A shared memory programming model for distributed memory computers. Proc. Supercomputing’94.
3. J. Nieplocha, B.Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, Proc. RTSPP IPPS/SDP’99, 1999.
4. S. Lumetta, A. M. Mainwaring, D.E. Culler, “Multi-Protocol Active Messages on a Cluster of SMP’s”. *Proc. Supercomputing’97*, 1997.

5. I. Foster, J. Geisler, C. Kesselman, S. Tuecke, “Managing Multiple Communication Methods in High-Performance Networked Computing Systems,” *Journal of Parallel and Distributed Computing,* Vol. 40, January 1997.
6. P. Husbands, J.C. Hoe, “MPI-StarT: Delivering Network Performance to Numerical Applications”. *Proc. Supercomputing’98*, 1998
7. Parallel Compiler Runtime Consortium, Common Runtime Support for High-Performance Parallel Languages, Proc. Supercomputing’93, 1993.
8. R. Govindaraju, IBM Power Parallel Systems, personal communication, 1999.
9. S.Andersson,G.Bhanot, J.Hague, F.Johnston, S. Kandalai, D. Klepacki, J. Levesque, J. Nieplocha, F. O’Connell, F. Parpia, C. Pospiech, *Scientific Applications in IBM RS/6000 SP Environments*, IBM Corp., ISBN: 0738415189, 1999.
10. M. Banikazemi, R. Govindraju, R. Blackmore, D. Panda, Implementing Efficient Implementation of MPI for IBM RS/6000 SP Systems, *Proc.* IPPS/SDP’99, 1999.
11. S.C. Woo, M.O. Ohara, E. Torrie, J.P. Singh, A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". *Proc. 22nd International Symposium on Computer Architecture*, 1995.
12. S.C. Woo, J.P. Singh, J.L. Hennessy, The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. *Proc.6th ASPLOS*, 1994.

A Comparison of Concurrent Programming and Cooperative Multithreading*

Takashi Ishihara, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson

Department of Computer Science, University of California, Davis, CA 95616 USA
{ishihara,liti,fodor,olsson}@cs.ucdavis.edu

Abstract. This paper presents a comparison of the cooperative multithreading models with the general concurrent programming model. It focuses on the execution time performance of a range of standard concurrent programming applications. The results show that in many, but not all, cases programs written in the cooperative multithreading model outperform those written in the general concurrent programming model. The contributions of this paper are an analysis of the performances of applications in the different models and an examination of the parallel cooperative multithreading programming style.

1 Introduction

The general concurrent programming execution model (CP) typically provides independent processes as its key abstraction. Processes execute nondeterministically. That is, processes run in some unknown order, which can vary from execution to execution, and context switches can occur arbitrarily. Multiple processes within a given program may execute at the same time on multiple processors, e.g., on a shared-memory multiprocessor or in a network of workstations. This model of execution is found in many concurrent programming languages — e.g., Ada [9], CSP [8], Java [5], Orca [3], and SR [1]. These languages provide various synchronization mechanisms to coordinate the execution of processes (e.g., semaphores, monitors, or rendezvous).

The cooperative multithreading execution model (CM) is a more specialized model of execution. Threads execute one at a time. A thread executes until it chooses to yield the processor or to wait for some event to become true. The kinds of events for which a thread can wait include a shared variable meeting a particular condition, a device completing some operation, or a timeout occurring. This model of execution is especially well-suited for writing programs for real-world programmable controllers for embedded systems [2], such as those found in irrigation control systems and railroad crossing control systems. One language for writing these controllers is Z-World's Dynamic C [15].

The CM model as defined above allows only one thread to be active at any given time. A natural generalization of CM (called PCM, for Parallel CM) is

* This work is partially supported by Z-World, Inc. and the University of California under the MICRO program.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 729–738, 2000.
© Springer-Verlag Berlin Heidelberg 2000

to allow multiple threads to be active simultaneously, so that a CM program can run on multiple processors. However, to preserve some of the advantages of CM (described later), some restrictions need to be placed on which particular threads can be run simultaneously. For example, only one thread per module, as in Lynx [12, 13], or one thread per group of threads with possibly interfering variable usages may be active at any time.

Two significant advantages of CM have been pointed out in the key work [12, 13] and specifically for controllers in [2]: CM is a simpler conceptual model and threads often do not need to synchronize explicitly because threads yield the processor at fixed places in the code. Two other tradeoffs [6] involve how the effect of I/O can vary in the different models and the relationship of execution fairness to program determinacy.

In this paper, we present a comparison of the cooperative multithreading models (CM or PCM) with the general concurrent programming model (CP). We focus on the execution time performance of a range of standard concurrent programming applications. The results show that in many, but not all, cases programs written in the CM- or PCM-style outperform those written in the CP-style. The contributions of this paper are an analysis of the performances of applications in the different models and an examination of the PCM programming style. (This paper extends our preliminary performance comparison of CM and CP [6].) Although the programs used in our experiments are written in SR, the general performance results should apply to other languages and systems. The specific performance results will vary depending on relative costs of synchronization and context switches, etc.

The rest of this paper is organized as follows. Section 2 briefly compares language features typical in the three models. Section 3 presents execution time performance comparisons of programs written in the CM- or PCM-style with their counterparts written in the CP-style for several standard CP applications. Section 4 addresses additional issues. Finally, Section 5 concludes the paper.

2 Language Features

We assume most readers are familiar with CP languages (such as Ada, etc. mentioned in Section 1) but are less familiar with CM or PCM languages. We therefore briefly present the essential ideas of two such languages — Dynamic C [15] and Lynx [12, 13].

Dynamic C extends the C language with various features to support CM. A thread executes until it chooses to yield the processor or to wait for some event to become true. Yielding the processor is accomplished via explicit statements: **yield** and **waitfor**. **yield** context switches to another ready thread, if any, or resumes the current thread if no other thread is ready. **waitfor** evaluates the condition. If true, the thread continues; otherwise, the thread yields and will, therefore, reevaluate the condition when it runs again. (Some notations use **await** rather than **waitfor**.)

Lynx is conceptually similar to Dynamic C. A Lynx program consists of a single module, called a *process*. Each process may contain multiple threads. As in Dynamic C, only one thread may be active at a time and threads execute until they block. An important feature of Lynx is that Lynx programs (processes) can communicate via messages using a *link* mechanism; the receipt of a message creates a new thread to run code to handle the message. Lynx falls under the PCM model because multiple threads — but at most one in each process within a group of processes — may execute at a time.

To illustrate the differences between the CP and CM models, Figure 1 shows

```
do true ->
   # think
     ...
   # get forks
   # (use semaphore operations
   # on shared sem array fork;
   # left and right are indices
   # of neighboring philosophers)
   P(fork[left]);P(fork[right])
   # eat
      ...
   # release forks
   V(fork[left]);V(fork[right])
od
```

(a) CP-style

```
do true ->
   # think
     ...
   # get forks, by simulating:
   #   await fork[left]=1 & fork[right]=1
   do not(fork[left]=1 & fork[right]=1)->
     nap(0) # i.e., yield
   od
   fork[left] := 0; fork[right] := 0
   # eat
      ...
   # release forks
   fork[left] := 1; fork[right] := 1
od
```

(b) CM-style

Fig. 1. Code for a Philosopher in Dining Philosophers

how the classic dining philosophers problem can be solved in CP and CM. The CP code uses standard SR features; for synchronization, it uses the shared array of semaphores `fork`. The CM code use only CM-like features from SR; for synchronization, it uses the shared array of integers `fork` and a simulated **await** statement. This simulation uses SR's `nap` function to explicitly yield. The CM code is compiled with an option that prevents normal, implicit context switches.

The key difference in these program fragments is how the philosopher checks the status of its two neighboring philosophers to decide whether it can eat. In CP, synchronization is required to avoid race conditions. By contrast, in CM, a context switch can occur only explicitly. Thus, no context switch can occur within evaluation of the conditions of the (simulated) **await** or between that test, if true, and the subsequent setting of the two elements of `fork`. (Note that the CM code here is not valid under PCM; see Section 3.2.)

3 Experimental Results

We wrote in the CM and PCM programming styles several standard CP applications including grid computations such as Jacobi Iteration (JI) (for approximating the solution to a partial differential equation), http servers, Producer/Consumer (PC[1]), Dining Philosophers (DP), Readers and Writers (RW), Matrix Multiplication (MM), and Traveling Salesman Problem (TSP). For the PC problem, we use the notation xPyCzS to mean x producers, y consumers, and a buffer with z slots.

We focus below on the DP, PC, and JI applications. We programmed the applications in the SR language, using standard CP features or CM-like features, as we did for the DP code in Figure 1. The PC and JI programs are fairly straightforward (the CP versions are taken from [1]).

Below, we compare CP with CM (both running on a single processor)[2] and CP with PCM (both running on a multiprocessor) by looking at the execution times for some of the applications mentioned above. We ran on many different problem sizes for each application. For example, for JI we tested with different size matrices, convergence values, and initial values. For PC, we tested with different numbers of consumer processes, producer processes, and number of slots in the buffer. For PC and DP, we tested with different amounts of time spent inside and outside of critical sections. The results we report below are representative of and summarize the observed results; see [10] for complete details.

To understand better where execution time was being spent, we modified the SR run-time system to report, upon program termination, the number of context switches, the total number of semaphore P operations performed, and the number of P operations that block. We also ran several micro-benchmarks to determine the basic costs of context switches and costs of P operations that block and of those that do not block. Finally, we used *gprof* to profile the code.

3.1 CP versus CM (Single Processor)

These tests were run on several workstations including a Sun Sparc 5 workstation running SunOS 5.5, various Intel-based PCs running various versions of Linux, a DEC 5000/240 running ULTRIX 4.3, and a DEC Alpha running OSF1 V3.2. The data presented in this section are from the tests run on the Sparc 5 or the Pentium Pro 200 running Linux 2.2.5-22. The overall patterns of results on all platforms are similar. We also focus on relative performance and so give most results in terms of the ratio of execution times multiplied by 100%, i.e., $T_{cm}/T_{cp} * 100\%$.

Table 1 shows representative results for the DP, PC (1P1C1S), and RW problems. The "work" in the table indicates how much non-critical activity a process

[1] The PC problem with more than one slot is usually called the Bounded Buffer problem. We use PC for both in this paper.

[2] CM applications are typically run on a single processor; some CP applications are run on a single processor.

WORK	DP	PC	RW
100	20.0	92.7	77.8
1000	44.3	92.9	74.8
10000	52.8	93.1	70.8
100000	53.8	93.3	69.8
1000000	52.9	93.1	70.2

Table 1. Execution time ratio CM/CP (%) for three applications

performs. For example, in PC, it indicates how much relative work a producer takes to produce a new item. We expect that most practical applications would fall within the lower work categories, which incur fewer context switches and synchronization points.[3]

For the applications in Table 1, the CM programs perform better for two reasons: they perform no P/V operations (i.e., they synchronize using shared variables and less often) and they make fewer context switches. Of these two factors, the former is more important because it is more costly. For example, on a Sparc-5, it takes 20-30 μs per P operation versus 5 μs per context switch.

These results prompted us to look further into how we could reduce costs. We observed that, in CM programs, processes were sometimes busy waiting more than necessary due to the order in which processes execute. For example, in the code for 2P2C1S, if a producer that has just deposited an item yields to the other producer, then that producer will be awakened, see that it cannot proceed, and in turn will yield. Some context switching could be eliminated if the producer instead yields to a consumer. We call this ability to select the process to which to yield a *named yield* and the usual yield an *unnamed yield*. (A named yield is like a coroutine resume.)

We simulated named yield for the PC problem. The graph in Figure 2 shows the results. The results are better than those shown for PC in Table 1. The results are also better on some tests for small amounts of work (*not* shown in Table 1), where the execution time ratios (%) using unnamed yields ranged from 90% to 120%.

We also looked at TSP, MM, and JI. For TSP and and MM, the programs written in the CM model obtain execution time ratios (%) between 97-99%, due to the small amount of synchronization in those applications. For JI, the CM model obtains execution time ratios (%) of between 98.5-100%, despite somewhat frequent barrier synchronization. For this application, the CM's barriers — busy waiting on shared variables with yields — turned out to cost about the same as the CP's barriers — performing P/V semaphore operations.

[3] Note that "work" means iterations of some computational loop. It does not necessarily correspond directly to elapsed time because, for example, if a process is waiting a long time for I/O to complete, other processes can execute during that time.

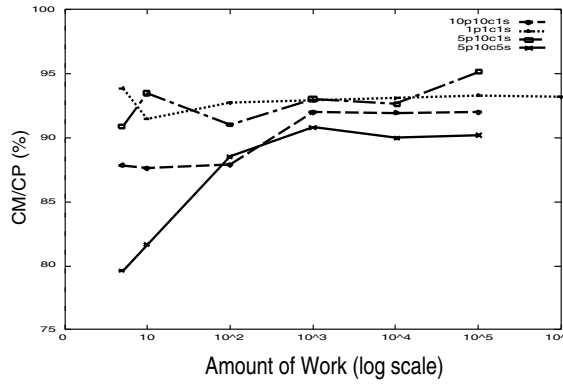


Fig. 2. Execution time ratio CM/CP (%) for the xPyCzS Problem

3.2 CP versus PCM (Multiprocessor)

We ran further experiments on a dual 550 MHZ processor PC running Red Hat Linux 6.0. These experiments used the SR implementation (MultiSR) with support for multiprocessors.

We ran the applications mentioned previously. Again, the results for TSP and MM differed little due to the small amount of synchronization in those programs. Note that the PCM versions do need to synchronize between groups of processes (more on that below). The more interesting tests were DP, PC, and JI.

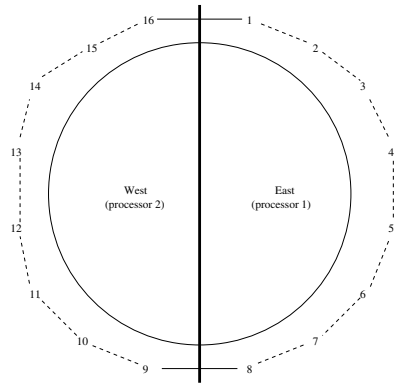


Fig. 3. Layout of PCM Dining Philosophers for 16 philosophers and 2 processors

For the DP problem, we used the same CP version as before (Figure 1(a)). The PCM version, though, is new. The basic approach, illustrated in Figure 3, splits philosophers into two regions: East and West. Each of these regions is

assigned to a processor. Synchronization within a region uses shared variables, represented by dashed lines in Figure 3. Synchronization between the two regions, however, uses a semaphore, represented by solid lines in Figure 3. The code is, therefore, a hybrid of the code in the two parts of Figure 1 with three kinds of philosophers: interior philosophers 2-7 and 10-15, each of which uses shared variables to get both of its forks; borders 1 and 9, each of which uses a semaphore to get its right fork but a shared variable to get its left fork; and borders 8 and 16, each of which uses a semaphore to get its left fork but a shared variable to get its right fork.

The MultiSR implementation, unfortunately, does not support processor affinity. (The same holds true for the underlying LinuxThreads on which MultiSR is built.) So, different processes in the same region (East or West) could run at the same time, which violates the PCM assumption and could lead to a race condition on the shared variables.

In our simulation, therefore, we tested two versions — DP_1 and DP_2 — of PCM DP. Both include extra (semaphore) synchronization to protect the shared variables ("protection synchronization"). DP_2 includes additional synchronization to ensure that only one process in a region runs at the same time ("region synchronization"). DP_1, on the other hand, allows more than one from the same region to run at the same time. DP_1 is a reasonable, conservative approximation to how PCM DP would perform. Given the characteristics of the tests (e.g., number of philosophers), it is likely that multiple philosophers from each region can run at the same time; so the overall performance is not likely to be improved by running two philosophers from the same region at the same time. The extra, protection synchronization means the measured costs are (most likely) higher than they would be for a pure PCM DP solution.

Table 2 shows some representative speed-ups for the two PCM programs compared with the CP DP program. DP_1 outperforms the CP version of DP for all tested workloads, even though DP_1 has extra, protection synchronization. DP_2 performs considerably worse for most workloads due to its use of extra, region synchronization.

WORK	DP_1	DP_2
2	45.4	57.9
4	54.6	83.0
10	81.8	126.0
20	86.6	139.1
100	99.7	178.2
200	95.0	171.9
400	98.1	172.2
1000	95.9	167.4

test	PC
1	101.5
2	102.8
3	101.8
4	100.8

test	JI
1	52.6
2	94.8
3	88.4
4	83.8

Table 2. Execution time ratio PCM/CP (%) for three applications

We tested a PCM version of the PC program. The synchronization requirements of the program led us to place all producers on one processor and all

consumers on the other because producers (consumers) need to synchronize in accessing the variable that indicates which item was removed (inserted). Unfortunately, the extra, protection synchronization required to protect shared variables results in code whose semaphore structure is identical to that in the CP version *and* has additional synchronization to simulate the **await** statements. Accordingly, its performance was slightly worse than the CP version, as indicated in Table 2.

The CP and PCM versions of JI each use a barrier. The difference, though, is in how the barrier is coded. In the CP version, the barrier uses semaphores shared by all processes. On the other hand, the PCM version is similar in spirit to the PCM version of DP (DP_1). The processes are divided into two groups. Each group of processes uses shared variables to implement the barrier among the group. Two semaphores are used to synchronize between the two groups. (The actual code uses the extra, protection synchronization, as in DP_1, to prevent race conditions.)

Table 2 shows some representative speed-ups for JI. As shown, the PCM version performs better, despite the extra, protection synchronization, than the CP version. The reason is its use of the simple shared variable barrier versus the more expensive semaphore based barrier. Note that the improvement for JI here is significant, whereas the improvement for JI in CM was nearly non-existent (see Section 3.1). The difference is that processes do fewer busy waits since they are now split onto two processors.

4 Discussion

Applications such as MM, TSP, and JI run on a uni-processor system will (almost always) be faster if they are written as sequential rather than concurrent programs. However, applications such as PC, DP, and RW represent servers. Those applications and others such as http or file servers lend themselves to multiple logical threads. For example, an http server multiplexes multiple connections and might have one thread for each request being serviced. These threads might be expressed as processes within a CP program or as threads within a CM program (e.g., as in the Boa server [4]). Having logical threads can also be important with respect to I/O [6].

The results we gave are based on measurements of SR programs, both standard, CP-style programs and CM- and PCM-style programs. The results might be skewed a bit in favor of the CP-style programs because SR's underlying runtime system was designed for CP. However, having all applications written in the same language was useful: the implementation of other language features (e.g., code generated for accessing array elements) is consistent between the different versions of programs and therefore does not unfairly influence the results as it might when comparing programs written in different languages.

The style in which programs are written differs between the models. The difference between CP and CM can be seen clearly in Figure 1. The difference between CP and PCM was described, for example, for the DP problem in Sec-

tion 3.2. There, the code is a hybrid of styles and the argument made regarding simplicity in [12, 13] is not as solid — the programmer needs to understand both models of execution to program in PCM.

The PCM DP example also raises the issue of load balancing. As presented, the processes were split statically into two groups (regions), under the implicit assumption that, overall, each group of processes was doing about the same amount of work. If that assumption is not correct, then philosophers could be moved. Specifically, a border philosopher could be moved from one group to the other; such a change would directly affect three philosophers and their roles as border or interior. The actual code to effect such a move would be complicated, especially when the philosophers are in the midst of synchronizing. In the CP model, load balancing can happen implicitly without any change to how the processes synchronize.

A potential advantage of PCM-style programs is that they might benefit from cache affinity [14]. That is, processes that use the same variables will be placed on the same processor, for example, as in the PCM DP example.

Our work is related generally to other work that attempts to eliminate synchronization or replace synchronization by less expensive forms. Examples: eliminating barrier synchronization from parallel programs [7] and replacing more costly forms of message passing with less costly ones [11]. Both of those approaches employ compiler analysis, whereas the approach in this paper is aimed at the higher, language level.

We are investigating how to transform CP programs into PCM or CM programs. Even if some programmers prefer to express their code within the CP model, that code can be transformed to PCM and run more efficiently by eliminating synchronization. For example, a P/V pair of semaphore operations used for mutual exclusion can simply be eliminated under certain conditions. Note that some implementations of CP languages essentially map a CP into a PCM program anyway, but they generally need to assume the worst case of when context switches will occur. It is also desirable to automatically transform programs like the CP version of DP into a PCM version. However, it is not clear how to devise general transformations that would work for many programs.

5 Conclusion

We have presented a comparison of the cooperative multithreading models (CM and PCM) with the general concurrent programming model (CP). We examined execution time performance of a range of standard concurrent programming applications. The results showed that in many, but not all, cases programs written in the CM- or PCM-style outperform those written in the CP-style. The key factor is that cooperative multithreading allows less costly synchronization to be used or even some synchronization to be eliminated. We also examined the PCM programming style. Our experience indicates that the cooperative multithreading models (CM or PCM) are viable alternatives to the general concurrent programming model (CP) and are worthy of further exploration.

Acknowledgement

Joel Baumert made valuable technical suggestions on this work. The anonymous reviewers provided detailed comments that helped us to improve this paper.

References

[1] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[2] Tak Auyeung. Cooperative multithreading. *Embedded Systems Programming*, pages 72–77, December 1995.

[3] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[4] `http://www.boa.org`, 1999.

[5] G. Cornell and C. S. Horstmann. *Core Java*. Sun Microsystems, Inc., Mountain View, CA, 1996.

[6] E.F. Fodor and R.A. Olsson. Cooperative multithreading: Experience with applications. In *The 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pages 1953–1957, July 1999.

[7] H. Han, C.-W. Tseng, and P. Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 25(5):591–612, October 1998.

[8] C.A.R. Hoare. Communicating Sequential Processes. *Communications ACM*, 21(8):666–677, August 1978.

[9] Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/Ada- IC/standards/95lrm_rat`.

[10] Takashi Ishihara, Tiejun Li, Eugene F. Fodor, and Ronald A. Olsson. Cooperative multitasking versus general concurrent programming: Measuring the overhead associated with synchronization mechanisms. Unpublished Manuscript, University of California, Davis, September 1999.

[11] C. M. McNamee. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming*, 19(5):357–387, October 1990.

[12] M. L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, 13(1):88–103, January 1987.

[13] M. L. Scott. The Lynx distributed programming language: Motivation, design and experience. *Computer Languages*, 16(3/4):209–233, 1991.

[14] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, pages 26–40, December 1991.

[15] Z-World, Inc. *Dynamic C 5.x Integrated C Development System Application Frameworks (Rev.1)*, 1998. Dynamic C 5.x.

The Multi-architecture Performance of the Parallel Functional Language GpH

Philip W. Trinder[1], Hans-Wolfgang Loidl[1], Ed. Barry Jr.[†], M. Kei Davis[2], Kevin Hammond[3], Ulrike Klusik[4], Simon L. Peyton Jones[5], and Álvaro J. Rebón Portillo[3]

[1] Heriot-Watt University, Edinburgh, U.K;
{trinder,hwloidl}@cee.hw.ac.uk
[2] Los Alamos National Laboratory, U.S.A;
kei@lanl.gov
[3] University of St. Andrews, U.K;
{kh,alvaro}@dcs.st-and.ac.uk
[4] Philipps–University Marburg, Germany;
klusik@mathematik.uni-marburg.de
[5] Microsoft Research Ltd, Cambridge, U.K;
simonpj@microsoft.com

Abstract. In principle, functional languages promise straightforward architecture-independent parallelism, because of their high level description of parallelism, dynamic management of parallelism and deterministic semantics. However, these language features come at the expense of a sophisticated compiler and/or runtime-system. The problem we address is whether such an elaborate system can deliver acceptable performance on a variety of parallel architectures. In particular we report performance measurements for the GUM runtime-system on eight parallel architectures, including massively parallel, distributed-memory, shared-memory and workstation networks.

1 Introduction

Parallel functional languages have several features that should, in theory, enable good performance on a range of platforms. They are typically only *semi-explicit* about parallelism, containing limited explicit control of parallel behaviour. Instead the compiler and runtime-system extract and exploit parallelism, with the programmer controlling a few key aspects of the parallelism explicitly. Purely functional languages also have *deterministic parallelism*: the value computed by a program is not dependent on its parallel behaviour, thereby avoiding the complications of race conditions and deadlocks. Many pure functional language implementations support *dynamic resource allocation*: the resources of the parallel machine are allocated during program execution. Dynamic resource allocation

[†] This paper is dedicated to the memory of Ed Barry Jr., who died an untimely death in May 1999.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 739–743, 2000.
© Springer-Verlag Berlin Heidelberg 2000

relieves the programmer from architecture dependent tasks such as specifying exactly what computations are to be executed where.

The cost of high-level, dynamically-managed parallelism is a complex compiler and/or runtime-system. Can such a sophisticated system deliver acceptable performance on very different parallel architectures? We tackle this question in the context of our GUM implementation of Glasgow Parallel Haskell (GpH), a non-strict functional language. GUM performs parallel graph reduction, and many aspects of the parallelism are determined dynamically e.g. threads are dynamically created and allocated to processors. GUM is designed to be portable, and uses a message-passing model (Sect. 2). Performance is measured for a simple test program with good parallel behaviour (Sect. 3) as well as for one larger application with irregular parallelism and complex data structures (Sect. 4). This complements our earlier research on parallelising substantial Haskell applications [1] and developing a suite of simulation and profiling tools.

2 The GUM Runtime System

GUM is the runtime-system for GpH [7], a parallel variant of the Haskell lazy functional language. Being a parallel graph reduction machine [3], GUM represents an architecture-independent abstract machine-model appropriate to both shared- and distributed-memory architectures. In this model both data and program are represented via graph structures. Executing a program means rewriting a graph with its result. Semi-explicit parallelism in GpH requires the programmer to annotate expressions that can be evaluated in parallel. The runtime-system then dynamically distributes data and work among the available processors. Potential parallelism may be subsumed [3] by existing threads in a similar way as in the lazy task creation mechanism [2], thereby dynamically increasing thread granularity. This dynamic granularity control, together with overlapping computation with communication (latency hiding), is crucial for achieving high performance on very different parallel architectures. Communication between threads is realised via a shared heap with implicit synchronisation on graph structures shared between several threads (implemented via message-passing on top of PVM or MPI). For efficient and portable compilation we use the Glasgow Haskell Compiler [4] (GHC), a state-of-the-art optimising compiler for Haskell. The design and implementation of GUM are discussed in detail in [7].

3 Measurement Setup

In our measurements we have used eight machine configurations: one MPP, a 97-processor Connection Machine CM-5 with the native CMMD communications library; one DMP, a 16-processor IBM SP/2 with MPI; one SMP, a 6-processor Sun SparcServer with PVM; a 56-node Beowulf cluster with 450MHz Intel Pentium II processors, 384MB RAM and 8GB local disk; and four networks of workstation (NOWs) all with PVM. The Beowulf uses a 100Mb/s fast Ethernet switch, the NOWs use standard Ethernet on the same subnet.

Table 1. Single-processor efficiency of `blackspots`

Class and Architecture	Sequential Runtime (s)	Efficiency	Class and Architecture	Sequential Runtime (s)	Efficiency
SMP			**Workstation-net**		
Sun-SMP PVM	135.2	77%	Digital Alpha PVM	378.6	63%
			Sun-4/15 PVM	815.4	84%
			Sun-10 PVM	289.1	96%
			Pentium PVM	109.4	96%
			Beowulf PVM	19.8	90%

In order to assess the overhead of the GUM runtime-system we have measured `parfact`, a simple binary divide-and-conquer program computing the sum of a given interval with little communication and no software bound for the achievable parallelism. The `parfact` program and additional details and measurements are available in [5]. The CM-5 achieves the best relative speedup of 74.1 on 97 processors, without approaching a parallelism bound imposed by either the hardware or the GUM runtime-system. These speedups are significantly better than the PVM and MPI versions, 5.82 and 6.53 on 8 processors, respectively, indicating the high costs of the portable communication libraries. Furthermore the MPI version on the IBM SP/2 suffers from a slow (sequential) startup. The Beowulf achieves a relative speedup of 33.1 on 56 processors. However, its parallel efficiency ($33.1/56 = 59\%$) is not as good as the CM-5's ($54.0/63 = 86\%$).

4 Accident Blackspots: A Larger GpH Program

The *Accident Blackspots* program determines locations where two or more traffic accidents have occurred, based on a set of police accident records as input. A number of criteria can be used to determine whether two accident reports are for the same location, and each criteria partitions the set. The problem amounts to combining several partitions of a set into a single partition, or *union find*. The program comprises 1,500 lines of Haskell code and additional details are available in [1]. The parallel (GpH) version of the algorithm uses a geometric partitioning of the input data into 32 small and 8 large tiles. Evaluation strategies [6] are used to define the parallel evaluation over these tiles.

The best sequential *efficiency* (see Table 1) is obtained for machines based on SPARC and Pentium processors, since these architectures are best supported by the GHC compiler itself. The simpler `parfact` program has higher efficiency in most cases [5]. In GUM sequential efficiency is dominated by the costs for managing potential parallelism (essentially adding a pointer to an array of possible tasks) and for locking graph structures representing work. The former, although cheap in itself, prohibits optimisations because it requires the graph structures to exist in the heap. The latter requires several instructions and, without the help of sharing analysis, increases with program size.

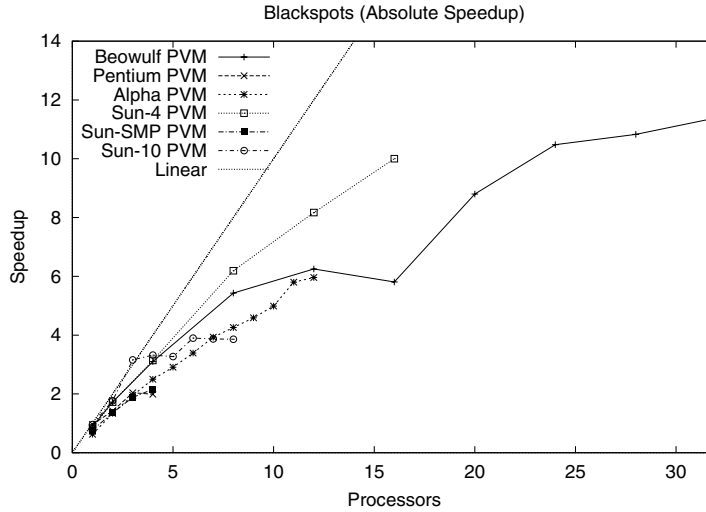


Fig. 1. Absolute speedups for `blackspots`

Fig. 1 shows that the small amount of communication required by the geometric partitioning enables good *speedups* even on the NOWs: 11.94 relative, 10.00 absolute on 16 Suns and 9.46 relative, 5.96 absolute on 12 Alphas. The Beowulf cluster profits from its high efficiency and its scalability. As a result it delivers the highest absolute speedup for this application: 11.39 (with larger input data 18.69) on 32 processors. Occasional drops in performance reveal that the dynamic scheduling is not always effective for this rather coarse-grained application. The overall poorer performance for the Pentium, Alpha and Sun-10 PVM NOWs is due to the higher ratio of communication costs to processor speed, exacerbated by the unusually low efficiency of the Digital Alpha. In this configuration the available parallelism is not sufficient to effectively hide communications latency. In the Beowulf cluster, with lower communications costs, this effect is less pronounced. In order to obtain good performance on machines with such characteristics we could perform architecture-dependent tuning, e.g. splitting the data into more, smaller tiles. The Sun-10 NOW exhibits a super-linear speedup for 3 processors. We believe this is due to reduced garbage collection costs in a parallel setting (with n processors we have n times the sequential heap available). The rather low speedup on the Sun SMP (2.82 relative, 2.16 absolute) is partly due to the higher overall performance of the processors and partly due to the competition with other user processes when performing the measurements. Our implementation would also profit from direct support of shared-memory — currently we use PVM or MPI even on SMPs.

5 Conclusion

In this paper we have assessed the architecture-independent performance of the GUM runtime-system for GpH by taking measurements on eight platforms,

ranging from MPPs to networks of workstations. From the `parfact` results we conclude that, for a program with little communication and no software bound on the parallelism, GUM is efficient on all platforms (at least 83%), capable of delivering acceptable speedups on all architectures, and also capable of massive parallelism (a relative speedup of 74 on a 97-processor CM-5). From the `blackspots` results we conclude that GUM can achieve absolute speedups (up to 18.69 on 32 processors) for symbolic programs with irregular parallelism. We consider GpH to be best suited for such symbolic applications, obtaining moderate speedups with only minimal code changes.

Our measurements suggest several improvements of GUM, which are currently being examined in the new implementation of the runtime-system. The implicit work and data distribution could be improved, e.g. by refining the work-stealing algorithm, by constructing clusters of data to avoid excessive communication or by supporting the migration of running threads. Furthermore, better interaction between granularity control and the generation of parallelism could be achieved via a low-watermark scheme maintaining a minimal amount of parallelism despite thread subsumption.

References

1. H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–752, Oct. 1999. Available from [8].
2. E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, Jul. 1991. <URL:`ftp://crl.dec.com/pub/DEC/CRL/tech-reports/90.7.ps.Z`>
3. S.L. Peyton Jones, C. Clack, and J. Salkild. High Performance Parallel Graph Reduction. In *Parallel Architectures and Languages Europe (PARLE'89)*, LNCS 365, pp. 193–206, Eindhoven, The Netherlands, Jun. 1989. Springer-Verlag.
4. S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, P.L. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Joint Framework for Information Technology Technical Conference*, pp. 249–257, Keele, U.K, Mar. 1993. See also <URL:`http://www.haskell.org/ghc`>
5. P.W. Trinder, Ed. Barry Jr., M.K. Davis, K. Hammond, S.B. Junaidu, U. Klusik, H-W. Loidl, S.L. Peyton Jones. Low Level Architecture-Independence of Glasgow Parallel Haskell (GpH). In *Glasgow Functional Programming Workshop*, draft proceedings, Pitlochry, Scotland, Sep. 1998. Available from [8].
6. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998. Available from [8].
7. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Programming Language Design and Implementation (PLDI'96)*, pp. 79–88, Philadelphia, PA, May 1996. Available from [8].
8. GPH Web Pages. <URL:`http://www.cee.hw.ac.uk/~dsg/gph`>

Novel Models for Or-Parallel Logic Programs: A Performance Analysis

Vítor Santos Costa[1], Ricardo Rocha[2], and Fernando Silva[2]

[1] COPPE Systems Engineering, Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br
[2] DCC-FC & LIACC, University of Porto, Portugal
{ricroc,fds}@ncc.up.pt

Abstract. One of the advantages of logic programming is the fact that it offers many sources of *implicit* parallelism, such as and-parallelism and or-parallelism. Arguably, or-parallel systems, such as Aurora and Muse, have been the most successful parallel logic programming systems so far. Or-parallel systems rely on techniques such as Environment Copying to address the problem that branches being explored in parallel may need to assign different bindings for the same shared variable. Recent research has led to two new binding representation approaches that also support independent and-parallelism: the Sparse Binding Array and the Copy-On-Write binding models. In this paper, we investigate whether these newer models are practical alternatives to copying for or-parallelism. We based our work on YapOr, an or-parallel copying system using the YAP Prolog engine, so that the three alternative systems share schedulers and the underlying engine.

1 Introduction

One of the advantages of logic programming (LP) is the fact that one can exploit *implicit* parallelism in logic programs. Implicit parallelism reduces the programmer effort required to express parallelism and to manage work. Logic programs have two major forms of implicit parallelism: *or-parallelism* (ORP) and *and-parallelism* (ANDP). Given an initial query to the logic programming system, ORP results from trying several different alternatives simultaneously. In contrast, ANDP stems from dividing the work required to solve the alternative between the different processors. One particularly interesting form of ANDP is independent and-parallelism (IAP), found in divide-and-conquer problems.

Arguably, ORP systems, such as Aurora [16] and Muse [2], have been the most successful parallel logic programming systems so far. One reason is the large number of logic programming applications that require search, including structured database querying, expert systems and knowledge discovery applications. Parallel search can be also useful in constraint logic programming.

Two major issues must be addressed to exploit ORP. First, one must address the *multiple bindings* problem. This problem arises because alternatives being exploited in parallel may give different values to variables in shared branches

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 744–753, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of the search tree. Several mechanisms have been proposed for addressing this problem [14]. Second, the ORP system itself must be able to divide work between processors. This *scheduling* problem is made complex by the dynamic nature of work in ORP systems.

Most modern parallel LP systems, including SICStus Prolog [5], Eclipse [1], and YAP [12] use copying as a solution to the multiple bindings problem. Copying was made popular by the Muse ORP system, a system derived from an early release of SICStus Prolog. The key idea for copying is that workers maintain separate stacks, but the stacks are in shared memory. Whenever a processor, say W_1, wants to give work to another, say W_2, W_1 essentially copies its own stacks to W_2. In contrast to other approaches, Muse [3] showed that copying has a low overhead over the corresponding sequential system. On the other hand, copying has a few drawbacks. First, it is expensive to exploit more than just ORP with copying, as the efficiency of copying largely depends on copying contiguous stacks, but this is difficult to guarantee in the presence of ANDP [15]. A second issue is that copying makes it more expensive to suspend branches during execution. This is a problem when implementing cuts and side-effects.

Recent research in the combination of IAP and ORP has led to two new binding representation approaches: the SBA (Sparse Binding Array) [8] and the αCOWL (copy-on-write design) [9]. The SBA is an evolution of Warren's Binding Array (BA) representation [20]. In BA systems, the stacks form a cactus-tree representing the search-tree, and processors expand tips of this tree. Workers thus use a shared pool of memory except when storing bindings that are private to a worker. These are stored in a local data-structure, the *Binding Array*. The αCOWL scheme uses a copy-on-write mechanism to do lazy copying. Both of these approaches elegantly support IAP and ORP.

The question remains of how these systems fare against copying for ORP, in order to verify whether they are indeed practical alternatives to copying. To address this question, we experimented with YapOr, an ORP copying system using the YAP engine [18], and we implemented the SBA and the αCOWL over the original system. The three alternative systems share schedulers and the underlying engine: they do only differ in their binding scheme. We then used a set of well known ORP all-solutions benchmarks to evaluate how they perform comparatively.

The paper is organised as follows. We first review in more detail the three models. Next, we discuss their implementation. We then present and discuss experimental results. Last, we make some concluding remarks.

2 Models for Or-Parallelism

A goal in our research is to develop a system capable of exploring implicitly all forms of parallelism in Prolog programs. A key point to achieve such a goal is to determine a binding model that simplifies the exploitation of the combined forms of parallelism. In this paper we concentrate in three binding models: environment copying, sparse binding arrays and copy-on-write. We assume a multisequencial

system, where the computational agents, or *workers*, do not initially inform the system that they have created new alternatives, and thus have exclusive access to these alternatives. This is called *private work*. At some point these alternatives may be made available to other workers, and we say they become *public work*.

The *Environment copying* model was introduced by Ali and Karlson in the Muse system [2]. In this model each computing agent (or worker) maintains a separate environment, almost as in sequential Prolog, in which the bindings it makes are independently recorded, hence solving the multiple bindings problem. When a worker becomes idle (that is, when it has no work), it searches for a busy worker from whom to request work. Sharing work among workers thus involves the actual copying of the computation state (WAM stacks) from the busy worker to the requester. After copying both workers have exactly the same state, and will diverge by executing alternative branches at the choice-point where the work sharing took place.

Efficient implementations of copying depend on *Incremental copying* to reduce the overheads of copying. With this technique, one just copies the parts of the execution stacks that are different among the workers involved. The scheduler plays an important role here by guiding idle workers to request work from the nearest busy workers. Bottom-most scheduling strategies have been very successful with this binding model, because they increase the number of choice-points shared between workers, thus preventing unnecessary copying.

The *Copy-On-Write* model, or αCOWL, was proposed by Santos Costa [9] towards supporting and/or parallelism. In the αCOWL, similarly to environment copying, each worker maintains a separate environment. Moreover, whenever a worker wants to share work from a different worker, it also *logically* copies all execution stacks. The insight here is that although stacks will be logically copied, they will be *physically* copied only on demand. To do so, the αCOWL applies the Copy-On-Write mechanism provided by most modern Operating Systems.

The αCOWL has two major advantages. First, we can copy anything. We can copy standard Prolog stacks, the store of a constraint solver, or a set of stacks for ANDP. Indeed, we might not even have a Prolog system at all. Second, because copying is done on demand, we do not need to worry about the overheads of copying large, non-contiguous, stacks. This is an important advantage for ANDP computations. The main drawback of the αCOWL is that the actual setting up of the COW mechanism can be itself quite expensive, and in fact, more expensive than just copying the stacks. In the next sections we discuss an implementation and its performance results.

The *Sparse Binding Array* (SBA) derives from Warren's Binding Arrays. Binding arrays were originally proposed for the SRI model [20]. In this model execution stacks are distributed over a shared address space, forming the so-called cactus-stack. In more detail, workers expand the stacks in the parts of the shared space they own, whilst they also can access stacks originally created by other workers. Note that the major source of updates to public and private work are bindings of variables. Bindings to the public part of the tree are tentative, and in fact different alternatives of the search tree may give different values,

or even no value, to the same variable. These bindings are called *conditional bindings*, and WAM-based systems will also stored them in the *Trail* data-area, so that they can later be undone. Conditional bindings cannot be stored in the shared tree. Instead, in the original BA scheme workers use a private array data structure associated with each computing agent to record conditional bindings.

The Sparse or Shadow Binding Array (SBA) [8] is a simplification of the BA designed to handle IAP. In the SBA each worker has a private virtual address space that fully shadows the system's shared address space. In other words, every work has its own shadow of the whole shared stacks. This "shadow" will be used to store to shared variables. Thus, the execution data structures and unconditional bindings are still stored in the shared address space, only conditional bindings are stored in the shadow area. The SBA thus preempts the problem of managing a BA in the presence of IAP [13], at the cost of having to allocate much more virtual space than the BA. A further optimisation in the SBA is that each SBA is mapped at the same fixed location for each worker in the system. We thus can maintain pointers from the shared stacks to the SBA.

3 Implementation Issues

The literature includes several comparisons of copying-based versus BA-based systems, and particularly of Aurora vs. Muse [4, 7]. One problem with these studies is that Aurora and Muse have very different implementations: it is quite difficult to know whether the differences stem from the model or from the actual implementation.

In contrast, we experimented with the three models by implementing them over the same YapOr system [18]. The system is derived from the Yap engine [10]. This is one of the fast emulator-based Prolog systems currently available, and only 2 to 3 times slower than systems that generate native-code. We would expect Yap to be between 2 to 4 times faster than the sequential Aurora engine on the same hardware.

3.1 YapOr with Copying

The YapOr system was originally designed to implement copying. The system is based on the Yap engine. The main changes were required on the instructions that manipulate choice-points. Other changes are in the initialisation code for memory allocation and worker creation, some small changes in the compiler to provide extra information for managing ORP, and lastly a change designed to support built-in synchronisation.

In a nutshell, the adapted engine communicates with YapOr through a fixed set of interface functions and through two special instructions. The functions are entered when choice-points are activated, updated, or removed. The two instructions are activated whenever a worker backtracks to the shared part of the tree and they call the *scheduler* to do work search. One instruction processes

parallel choice-points, and the other sequential choice-points (that is, choice-points such that their alternatives must be explored in sequential order).

The scheduler is the major component of YapOr. Work is represented as a set of *or-frames* in a special shared area. Idle workers consult this area and the `GLOBAL` and `LOCAL` data structures, which contains data on work and the status of each worker, until they find work. If there is no work in the shared tree, idle workers try to share work with a *busy* worker. This sharing is implemented by two model dependent functions: `p_share_work()`, for the busy worker, and `q_share_work()`, for the idle one. After sharing the previously idle worker will backtrack to a newly shared choice-point, whereas the previously busy worker continues execution from the same point. Note that before sharing, workers will try to move up in the tree to simplify incremental copying. In copying, sharing is implemented by the following algorithm:

```
        Busy Worker P            Signals            Idle Worker Q
-------------------------- ---------------- -------------------------
Compute stacks to copy                      Wait sharing signal
.                          ----sharing----> .
Share private nodes                         Copy trail ?
.                                           Copy heap ?
.                                           Wait nodes_shared signal
.                          --nodes_shared-> .
Help Q in copy ?                            Copy local stack ?
.                          <---copy_done--- .
.                          ---copy_done---> .
Wait copy_done signal                       Wait copy_done signal
Back to Prolog execution                    Install conditionals
.                          <-----ready----- .
Backtrack to shared node ?                  Fail to top shared node
Wait ready signal                           .
```

Initially, the idle worker waits for a sharing signal while the busy worker computes the stacks to copy. Next, the busy worker prepares its private nodes for sharing whilst the idle worker performs incremental copying. The busy worker may help in the copying process to speed it up. The two workers then synchronise to determine the end of copying. At the end, the busy worker goes back to Prolog execution and the idle worker installs the conditional bindings from the busy worker that correspond to variables in the maintained part of the stacks. To guarantee the correctness of the installation step, the busy worker can not backtrack to a shared node until the idle worker do not completes installation.

3.2 αCOWL

To support sharing of work in the αCOWL we had to change `p_share_work()` and `q_share_work()`. We applied the main function where Unix-style Operating Systems implement COW: the `fork()` function. The idea is that whenever a worker `P` accepts a work request from another worker `Q`, worker `P` forks a child process that will assume the identity of worker `Q`, whilst the older process executing `Q` exits. At this point, the new process `Q` has the same state as that of `P`. The process `Q` then is forced to backtrack to the same choice-point. Note that

scheduling is realised in exactly the same way as for the environment copying model, that is through the use of a public tree of or-frames in shared space.

The synchronisation algorithm for sharing work in αCOWL is as follows:

```
       Busy Worker P            Signals            Idle Worker Q
-------------------------- ---------------- --------------------------
.                                           Wait sharing signal
.                          ----sharing----> .
fork()                                      exit()
.                                           Child takes Q's id
Back to Prolog execution                    Fail to top shared node
```

Note that `fork()` is a rather expensive operation. For programs which have parallelism of high granularity, one expects that the workers will be busy most of the time and the number of sharing operations be small. In this case the model is expected to be efficient. On the other hand, we would expect worse results for fine-grained applications. Note that one could use the `mmap()` primitive as an alternative to `fork()`, but we felt `fork()` provided the most elegant solution.

3.3 Sparse Binding Arrays

Supporting the SBA requires changes to both the engine and the sharing mechanism. The main changes to the engine affect pointer comparison, and variable and binding representation.

As regards *pointer comparison*, the Yap system assumes pointers in the stacks follow a well-defined ordering: the local stack is above the global stack, the local stack grows downwards, and the global stack grows upwards. These invariants allows one to easily calculate variable age and are useful for trailing and recovering space. Unfortunately, they are not valid in the SBA, as the cactus stack is fragmented. Aurora uses the BA offset as a means for calculating age, but has to pay the overhead of maintaining an extra counter. The Aurora/SBA implementation used an age counter that records the number of choice-points above the current choice-point [8]. The YAP SBA implementation does not maintain such counters, and instead follows the rule:

1. the sequential invariant is guaranteed to hold for private data;
2. shared data in the cactus-stack are protected as regards recovering space, and age follows the simple rule: smaller is older.

To implement this rule, each worker manages the so-called *frozen* registers that separate its private from the shared parts of the tree. Moreover, an extra register, *BB*, replaces the WAM's *B* register when detecting whether a binding is conditional. Note that these same problems must be addressed to support IAP.

The second issue we had to address is *variable representation*. In the original WAM a variable is represented as a pointer to itself. This is unfortunate, because we would need to initialise the whole of the BA. BA based systems (with the exception of Andorra-I [11]) thus assume unbound variables are ultimately null cells. In Aurora, a new variable is initialised as a tagged pointer to the BA, itself null. In the SBA we do not need pointers to the BA, as it is sufficient to

calculate the offset we are at in the shared space, and add it to the SBA base. Aurora/SBA thus initialises a new variable as a tagged age field.

We decided to optimise for the sequential overhead in the YapOr/SBA implementation. To do so, a new cell is initialised as a null field. Moreover, and in contrast to previous BA-based systems, conditional bindings will only be moved to the SBA *when they are made public, and only then.* This means that private execution in our scheme will not use the SBA at all.

As bindings are made public they will be copied to the SBA. Moreover, the original cell will be made to point to the SBA. Thus the variable dereferencing mechanism is unaware of the existence of the SBA. Note that the pointer that is placed in the original cell is independent of workers, although it points at a private structure.

The changes to the engine are therefore quite extensive. As regards the changes to `p_share_work()` and `q_share_work()`, the new algorithm is as follows:

```
        Busy Worker P            Signals            Idle Worker Q
-------------------------- ---------------- --------------------------
Compute stacks to share                     Wait sharing signal
Share private nodes                         .
.                          ----sharing----> .
.                                           Install conditionals
Back to Prolog execution                    Fail to top shared node
```

4 Performance Evaluation

In order to compare the performance of these three models we experimented the 3 systems in two parallel architectures: a Sun SparcCenter2000 with 8 CPUs and 256MB of memory, running Solaris2.7, and a PC server with 4 PentiumPro CPUs/200MHz/256KB caches and 128MB of memory, running Linux2.2.5 from standard RedHat6.0. Each CPU in the PC server is about 4 times as fast as each CPU in the SparcCenter. All systems used the same compilation flags.

We used a standard set of all-solutions benchmarks, widely used to compare ORP logic-programming systems [19]. We preferred all-solutions benchmarks because they are not susceptible to speculative execution, and our goal was to compare the models. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sorting resolution. Table 1 shows the execution time, in seconds, for Yap Prolog and the overhead, in percentage over Yap Prolog, introduced by each or-parallel model when executing with one worker.

The overhead for copying (YapOr) confirmed previous results, and is of the order of 2% or 13% on PC/Linux and Sparc/Solaris, respectively. The overhead obtained for αCOWL is equivalent to copying. The results are consistent, and the variations are quite above the noise in our measures. We expected performance to be about the same, as for a single processor we execute quite the same code: the systems only differ in their scheduling code, and this is never activated.

The overhead for SBA is, as expected, higher but not very much so, only of 15% or 32%. We believe this good result stems from the optimisations discussed

Programs	Yap Prolog		YapOr		αCOWL		SBA	
	PC	Sparc	PC	Sparc	PC	Sparc	PC	Sparc
cubes5	0.216	0.753	2%	13%	4%	10%	9%	21%
cubes7	2.505	9.042	1%	7%	3%	5%	7%	17%
ham	0.435	1.537	4%	20%	4%	24%	25%	57%
nsort	34.810	142.161	1%	14%	1%	21%	22%	55%
puzzle	2.145	8.411	2%	22%	2%	18%	22%	39%
queens10	0.703	2.809	3%	8%	4%	7%	12%	17%
queens12	20.921	84.600	2%	4%	3%	4%	10%	15%
Average			2%	13%	3%	13%	15%	32%

Table 1. Overheads Yap Prolog/Or-Parallel Models with one worker.

in the previous section. In fact, SBA vs. YapOr performs relatively better than Aurora vs. Muse. We believe this result supports continuing research on the SBA.

Table 2 shows speedups for the PC Server, and Table 3 for the SparcCenter. We use **Copy** for copying and **COW** for the αCOWL. The results show that the best speedups are obtained with copying. The SBA follows quite closely, but the speedups are not as good for higher number of workers. We believe this is partly a problem with the SBA optimisations. As work becomes more fine-grained, more bindings need to be stored in the Binding Array. Execution thus slows down as the system needs to follow longer memory references and touches more cache-lines and pages.

Programs	2 workers			3 workers			4 workers		
	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA
cubes5	1.99	1.88	1.98	2.97	2.60	2.97	3.95	2.69	3.98
cubes7	1.99	1.98	1.99	2.99	2.92	2.99	3.99	3.83	3.98
ham	1.97	1.90	1.99	2.93	2.64	2.97	3.82	2.79	3.87
nsort	2.03	2.00	1.96	3.06	2.98	2.93	4.08	3.90	3.90
puzzle	1.97	1.93	1.95	2.96	2.41	2.92	3.94	3.20	3.88
queens10	1.99	1.88	1.99	2.96	2.12	2.97	3.92	2.42	3.92
queens12	2.00	1.99	1.99	3.00	2.86	2.99	4.00	3.82	3.98
Average	1.99	1.94	1.98	2.98	2.65	2.96	3.96	3.24	3.93

Table 2. Speedups for the three models on the PC Server.

The results for the αCOWL are quite good, considering the very simple approach we use to share work. The αCOWL performs well for smaller number of processors and for coarse-grained applications. As granularity decreases the overhead of the `fork()` operation becomes more costly, and in general system performance decreases versus other systems. As implemented, the αCOWL is therefore of interest for parallel workstations or for applications with large running times, which are indeed the ultimate goal for our work.

Programs	2 workers			4 workers			6 workers			8 workers		
	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA	Copy	COW	SBA
cubes5	2.00	1.83	1.96	3.95	2.72	3.70	5.79	2.87	4.88	7.35	2.32	6.02
cubes7	2.01	1.97	1.99	3.98	3.79	3.87	5.97	5.03	5.53	7.74	5.87	7.40
ham	1.98	1.78	1.90	3.79	2.54	3.98	5.57	3.00	5.33	6.97	2.15	7.29
nsort	1.94	1.97	2.02	3.83	3.77	4.01	5.69	5.42	5.88	7.42	6.03	7.77
puzzle	2.02	1.92	1.91	3.94	3.08	3.64	5.91	3.79	5.12	7.68	3.79	7.08
queens10	2.03	1.85	1.93	3.97	2.36	3.82	5.83	2.73	5.48	7.47	2.43	6.95
queens12	2.01	1.95	1.97	4.01	3.74	3.92	5.97	5.13	5.89	7.77	5.81	7.66
Average	2.00	1.90	1.95	3.92	3.14	3.85	5.82	4.00	5.44	7.49	4.06	7.14

Table 3. Speedups for the three models on the SparcCenter.

5 Conclusions

We have discussed the performance of 3 models for the exploitation of ORP in logic programs. Our results show that copying has a somewhat better performance for all-solution search problems. The results confirm the relatively low overheads of copying for ORP systems.

Our results confirm that the SBA is a valid alternative to copying. Although the SBA is slightly slower than copying and cannot achieve as good speedups, it is an interesting alternative for the applications where copying does not work so well. As an example we are using the SBA to implement IAP.

Our implementation of the αCOWL shows good base performance, but suffers heavily as parallelism becomes more fine-grained. Still, we see the αCOWL as a valid alternative for the good reason that the applications that interest us the most have very good parallelism. The αCOWL has two interesting advantages for such applications: it facilitates support of extensions to Prolog, such as sophisticated constraint systems, and it largely simplifies the implementation of garbage collection, that in this model can be performed independently by each worker. The next major challenge for the αCOWL will be the support of suspension, required for single-solution applications.

We would like to perform low-level simulation in order to better quantify how the memory footprints and miss-rates differs between models. Work on these models is progressing apace. We are working on better application support for constraint and inductive logic programming systems. Moreover, we are using copying as the basis for parallelising tabling [17], useful say for model-checking, and the SBA as the basis for IAP [6], that has been used in natural language applications.

Acknowledgments

The authors would like to acknowledge and thank the contribution and support from Eduardo Correia. The work has also benefitted from discussions with Luís Fernando Castro, Inês de Castro Dutra, Kish Shen, Gopal Gupta, and Enrico

Pontelli. Our work has been partly supported by Fundação da Ciência e Tecnologia and JNICT under the project Dolphin (PRAXIS/2/2.1/TIT/1577/95) and by Brazil's CNPq under the NSF-CNPq project $CLoP^n$.

References

[1] A. Aggoun and et. al. *ECLiPSe 3.5 User Manual*. ECRC, December 1995.
[2] K. A. M. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.
[3] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *NACLP'90*, pages 757–776. MIT Press, October 1990.
[4] A. Beaumont, S. M. Raman, P. Szeredi, and D. H. D. Warren. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE'91*, volume 2, pages 403–420. Springer Verlag, June 1991.
[5] M. Carlsson and J. Widen. SICStus Prolog User's Manual. SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.
[6] L. F. Castro, V. S. Costa, C. Geyer, F. Silva, P. Kayser, and M. E. Correia. DAOS: Distributed And-Or in Scalable Systems. In *EuroPar'99*. Springer-Verlag, LNCS, August 1999.
[7] M. E. Correia, F. Silva, and V. S. Costa. Aurora vs. Muse; A Performance Study of Two Or-Parallel Prolog Systems. *Computing Systems in Engineering*, 6(4/5):345–349, 1995.
[8] M. E. Correia, F. Silva, and V. S. Costa. The SBA: Exploiting Orthogonality in AND-OR Parallel Systems. In *ILPS'97*, pages 117–131. The MIT Press, 1997.
[9] V. S. Costa. COWL: Copy-On-Write for Logic Programs. In *IPPS'99*. IEEE Press, May 1999.
[10] V. S. Costa. Optimising Bytecode Emulation for Prolog. In *PPDP'99*. Springer-Verlag, LNCS, September 1999.
[11] V. S. Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *ICLP'91*, 1991.
[12] L. Damas, V. S. Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1998. `http://www.ncc.up.pt/~vsc/Yap`.
[13] G. Gupta and V. S. Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *PARLE'92*, pages 617–632. Springer-Verlag, LNCS 605, June 1992.
[14] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM TOPLAS*, 15(4):659–680, 1993.
[15] Gopal Gupta, M. Hermenegildo, E. Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.
[16] E. Lusk and et. al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.
[17] R. Rocha, F. Silva, and V. S. Costa. Or-Parallelism within Tabling. In *PADL'99*, pages 137–151. Springer-Verlag, LNCS 1551, January 1999.
[18] R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *EPIA'99*, pages 178–192. Springer-Verlag, LNAI 1695, September 1999.
[19] P. Szeredi. Performance Analysis of the Aurora Or-parallel Prolog System. In *NACLP'89*, pages 713–732. MIT Press, October 1989.
[20] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *SLP'87*, pages 92–102, 1987.

Executable Specification Language for Parallel Symbolic Computation*

Alexander B. Godlevsky and Ladislav Hluchý

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 842 37 Bratislava, Slovakia
{godlevsky.ui,hluchy.ui}@savba.sk

1 Introduction

Two goals, simplicity of program designing, and efficiency of its computation, always remain topical in programming, and more than anything this is true about parallel programming systems. The former goal is usually achieved for declarative programming languages, the latter - by embedding of coordination level operators. One of the earliest such extensions, future annotation, isa proposed in [3]. Their use allows to start a function computation before the moment when computation of its annotated arguments will be completed. Another advance to increase program parallelization was using of nondeterministic operators in pseudo-functional languages [5]. One more resource widely used in logic programming to program parallelization is speculative computation of alternative branches.

In this paper we propose SL specification language that combines all of these features: 1) future annotating; 2) speculative annotating to point out conditional operators whose alternatives can be started to compute before the condition value will be determined; 3) sets as data structures and a nondeterministic choice operator with erratic choice semantics [6]. Such nondeterminism allows to choose elements from a set even if there are other set's elements that are in progress.

Another pecularity of our approach is SL program transformations during compile time stage illustrated below by an example.

2 SL Language, Its Sequential and Parallel Semantics

The syntax of SL language is presented below in a normalized let-form. Symbols **?** and **!** are used for future and speculative annotations, unite operation for sets union, $\mathsf{scar}(A)$ for nondeterministic choice element from A set, and $\mathsf{scdr}(A)$ defines the complement to $\mathsf{scar}(A)$ in A set. Though the language is not a functional one, we use the λ symbol to connect variables, similar to λ-calculi.

$$
\begin{aligned}
T ::= {} & x \mid (\mathsf{let}\ (x\ V)\ T) \mid (\mathsf{let}\ (x\ (\mathbf{?}\ T))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{car}\ y))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{cdr}\ y))\ T) \\
& \mid (\mathsf{let}\ (x\ (\mathsf{scar}\ y))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{scdr}\ y))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{unite}\ V\ V))\ T) \\
& \mid (\mathsf{let}\ (x\ (\mathsf{if}\ T\ T\ T))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{if}\ (\mathbf{!}\ T)\ T\ T)))\ T) \mid (\mathsf{let}\ (x\ (\mathsf{apply}\ y\ z))\ T)
\end{aligned}
$$

* This work was supported by the Slovak Scientific Grant Agency within Research Project No.2/7186/20

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 754–757, 2000.
© Springer-Verlag Berlin Heidelberg 2000

$V \in Value ::= c \mid x \mid (\ \lambda x.T) \mid (\mathsf{cons}\ x\ y) \mid S$ $\qquad x \in Vars ::= x, y, z, \ldots$

$c \in Const ::= \mathsf{true}, \mathsf{false}, \emptyset \ldots$ $\qquad S \in Set ::= \oslash \mid \{V\} \mid (\cup\ S\ S\ )$

The sequential semantics of SL is the relation $eval_{seq}$ which corresponds a set of permissible results to a closed program where the result is either a closed value whose all λ-expressions are replaced by procedures, or error indicating that an operation was misapplied, or $\perp$ if program is not terminated. This semantics is call-by-value, and it is defined by sequential SSL-machine which cannot be decribed here because of lack space. Transition rules for **?** and **!** annotations define them as identity operations requiring that the expression which they are applied to will be reduced first to a value and only then the expression's name will be replaced by the value. All transition rules correspond to common knowledge of the language constructors.

The parallel semantics is defined by parallel PSL-machine whose transition rules are partially presented in Fig.1. With reference to the SSL-machine, its state space uses an additional class of values - *placeholder* variables. Likewise to functional language with future annotations, these variables represent computation results still undeterminated. When one of parallel processes requires the value of an expression and this value is a placeholder corresponding to computation in another process, then the former process will be suspended. When the latter process is terminated, the placeholder will be replaced by the value computed and computation of the former process can go on.

Specification

$Q \in State_p \ ::= T \mid \mathsf{error} \mid (\mathsf{f\text{-}let}\ (p\ Q)\ Q) \mid (\mathsf{if\text{-}let}\ (p\ (p\ Q\ Q))\ Q)$ (States)

$p \in Ph\text{-}Vars ::= \{p_1, p_2, p_3, \ldots\}$ (Placeholders)

Transition Rules

$$\mathcal{E}[\,\mathsf{let}\ (x\ (\mathsf{if}\ !(T_0)\ T_1\ T_2))\ T] \rightarrow_{par} (\mathsf{if\text{-}let}\ T_0\ (\mathsf{let}\ (x\ T_1)\ \mathcal{E}(T))\ (\mathsf{let}\ (x\ T_2)\ \mathcal{E}(T))) \quad \exists p.\, p \in \mathrm{FP}(T_0) \qquad \text{(spec)}$$

$$(\mathsf{if\text{-}let}\ T_0\ Q_1\ Q_2) \rightarrow_{par} (\mathsf{if\text{-}let}\ T_1\ Q_3\ Q_4) \quad \text{if } T_0 \rightarrow_{par} T_1,\ Q_1 \rightarrow_{par} Q_3,\ Q_2 \rightarrow_{par} Q_4 \qquad \text{(par spec)}$$

$$(\mathsf{if\text{-}let}\ (V\ Q_1\ Q_2)) \rightarrow_{par} \begin{cases} Q_1 & \text{if } V = \mathsf{true} \\ \mathsf{error} & \text{if } V = \mathsf{error} \\ Q_2 & \text{if } V \not\models \mathsf{true},\ p \end{cases} \qquad \text{(end spec)}$$

$$\mathcal{E}[\,\mathsf{let}\ (x\ (?\ T_1))\ T] \rightarrow_{par} (\mathsf{f\text{-}let}\ (p\ T_1)\ \mathcal{E}[\,T[x \leftarrow p]\,]), \quad p \notin \mathrm{FP}(T) \qquad \text{(future)}$$

$$(\mathsf{f\text{-}let}\ (p\ V)\ Q) \rightarrow_{par} Q[p \leftarrow V] \qquad \text{(join)}$$

$$(\mathsf{f\text{-}let}\ (p\ \ \mathsf{error})\ Q) \rightarrow_{par} \mathsf{error} \qquad \text{(error)}$$

$$(\mathsf{f\text{-}let}\ (p_1\ Q_1)\ Q_2) \rightarrow_{par} (\mathsf{f\text{-}let}\ (p_1\ Q_3)\ Q_4) \text{ if } Q_1 \rightarrow_{par} Q_3,\ Q_2 \rightarrow_{par} Q_4 \qquad \text{(par future)}$$

$$(\mathsf{f\text{-}let}\ (p_2\ \mathsf{f\text{-}let}\ (p_1\ Q_1)\ Q_2))\ Q_3) \rightarrow_{par} (\mathsf{f\text{-}let}\ (p_1\ Q_1)(\mathsf{f\text{-}let}\ (p_2\ Q_2)\ Q_3)) \quad p_1 \notin \mathrm{FP}(Q_3) \qquad \text{(lift)}$$

Fig. 1. Nondeterministic parallel PSL-machine

f-let and if-let are states of PSL-machine added to the SSL-machine states to support its parallel computation. Q_1 from (f-let $(p\ Q_1)\ Q_2$), the right hand side expression of *future* rule, is the body of **?**-expression, and Q_2 is its evaluation context, p placeholder represents the result of Q_1 computation in Q_2. FP function returns the set of all placeholders of the expression to which it is applied. According to sequential semantics of SL the computation of Q_1 is mandatory, and Q_2 computation is speculative since the result of the latter

can be unnecessary if the former remained nonterminated or its result is error. (if-let T_0 Q_1 Q_2) state is generated by (*spec*) rule for speculative computation along both if-operator branches. Q_1 and Q_2 expressions represent then- and else-branches, T_0 corresponds to the operator's condition that can be computed in parallel with Q_1, Q_2 expressions, but the only mandatory computation among them is running for the Q_0 condition. if-let state will be reduced according to (*end spec*) rule after Q_0 will be replaced by a proper value. (*spec*) and (*future*) rules initiate new parallel computations, (*par spec*) and (*par future*) run them, (*lift*) rule restructures of nested parallel states to increase the level of parallelism.

We define the correctness of parallel semantics with respect to sequential one as *each result possible for PSL-machine is also possible for SL-machine*, that is as the inclusion $eval_{par} \subseteq eval_{seq}$. Its proof is based on comparative behaviours of SSL- and PSL-machines.

3 Compile-Time Transformations

Together with the language and its semantics declaration, the key element of the paper are program transformations for compile-time stage. They allows modification of original equations with reduction of some intermediately generated speculative branches to a new compact form.

For instance, let us consider an SL-program $Gr(A, B)$ that (being without its annotations) defines the outer recursion of Buchberger's algorithm [1] of polynomial bases constructing. The condition to stop computation of Gr is $Gr(\emptyset, B) = B$ and its recursion is defined as follows

$$\begin{aligned} Gr(A,B) = {} & (\mathsf{let}\ f\ \ \mathbf{?}Red(\mathsf{scar}(A),B))\ \mathsf{if}\ \mathbf{!}(f{=}0) \\ & \mathsf{then}\ Gr(\mathsf{scdr}(A), B) \\ & \mathsf{else}\ Gr(\mathsf{unite}(\mathbf{?}Create(f,B),\ Filter(!(Div f),\ \mathsf{scdr}(A))),\ \mathsf{unite}(\{f\},\ B)). \end{aligned}$$

According to (*future*) and (*spec*) rules of PSL-machine the right-hand side of this recursive equation can be transformed to

$$\begin{aligned} & (\mathsf{f\text{-}let}\ ph_1\ \ Red(\mathsf{scar}(A),B))\ (\mathsf{if\text{-}let}\ (ph_1{=}0)\ [\mathsf{then}]\ Gr(\mathsf{scdr}(A), B) \\ & [\mathsf{else}]\ Gr(\mathsf{unite}(\mathbf{?}Create(ph_1, B), Filter(\mathbf{!}(Div\, ph_1),\ \mathsf{scdr}(A))),\ \mathsf{unite}(\{ph_1\},B))). \end{aligned}$$

Here then and else are comments for branches which can be computed in parallel. Our *trick* is to unify these branches using some identities and redefining (generating) some equations. In this case we use the identities $unite(\emptyset, A){=}A$ and $Filter(\mathsf{true},A){=}A$, and also generate the operators

$$\begin{aligned} Create1(x,y) &= (\ \mathsf{if}\ x{=}0\ \mathsf{then}\ \emptyset\ \mathsf{else}\ Create(x,y)\ ), \\ zero(x) &= (\ \mathsf{if}\ x{=}0\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \{x\}\ ). \end{aligned}$$

After such unification, both branches are equal to $Gr(\mathsf{unite}(\mathbf{?}Create1(ph_1, B),\ Filter(\mathbf{!}(Div\, ph_1),\ \mathsf{scdr}(A))),\ \mathsf{unite}(\mathbf{?}zero(ph_1),\ B))$. Now, eliminating the if-let branching and applying (*future*) rule for **?** annotation, the right-hand side of the Gr equation can be transformed as follows:

$$(\mathsf{f\text{-}let}\ ph_1\ \ Red(...))(\mathsf{f\text{-}let}\ ph_2\ \ Create1(ph_1, B))(\mathsf{f\text{-}let}\ ph_3\ \ zero(ph_1))$$
$$Gr(\mathsf{unite}(ph_2,\ Filter(!(Div\ \ ph_1),\ \mathsf{scdr}(A))),\ \mathsf{unite}(ph_3, B)).$$

To apply this equation on-line, it is necessary to replace the fixed names ph_1, ph_2, ph_3 by calls of a generator of such new names.

When the first parameter of Gr equation is not a set but an exrpession similar to $\mathsf{unite}(ph_2,\ Filter(\mathbf{!}(Div\ \ ph_1),A))$, the modification of this equation for next applications are more complicated. In this case we use the identities $\mathsf{scar}(\mathsf{unite}(ph, A)) = \mathsf{scar}(A)$, $\mathsf{scdr}(\mathsf{unite}(ph,\ A)) = \mathsf{unite}(ph,\ \mathsf{scdr}(A))$ and unfolding for $Filter$ operator. However, here we are limited to demonstrate these transformations.

4 Conclusion and Future Works

We consider annotations as a tool to coordinate computation in declarative programs. Nondeterministic choice operators, on the one hand, allow to balance the tendence to advance parallel computation even when some other processes are in progress, and, on the other hand, the tendence to be close to some optimal heuristics when choosing elements from sets.

One of the key elements of our approach are compile-time program transformations. These transformations are similar to ones from the theory of partial evaluations [4] in the sense that we consider placeholder variables as dynamical ones. Then, we use identities a part of which contain high-order functions (filter, f-let,...), and here we have elements close to program parallelization used in the skeleton theory [2]. Finally, some of our transformations (for example, unfolding) have to be applied in a lazy manner, and here we are interested in a parallelization experience of programs written in lazy functional languages [7].

References

1. B.Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of Zero-Dimensional Polynomial Ideal.* Ph.D.Thesis, Math.Inst., Univ. of Innsbruck, Austria, 1965.
2. M.Cole. *Algorithmic skeletons: structured Management of Parallel of Parallel Computation.* MIT Press, Cambridge, Mass., 1989.
3. R. Halstead. *Multilisp: A language for concurrent symbolic computation*, ACM Transaction on Programming Languages and Systems, **7**, 4 (1985), pp.501-538
4. N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, June 1993. xii + 415 pages.
5. Wolfgang Schreiner: *A Para-Functional Programming Interface for a Parallel Computer Algebra Package*, J. Symbolic Computation (1996), **21**, No 4, pp.593-614
6. H. Sondergard and P. Sestoft.*Non-determinism in functional languages*, The Computer Journal, **35**(5): 514–523, 1992.
7. Trinder , P., Hammond, K., Loidl, H.-W., Peyton Jones, S. *Algorithm + Strategy = Parallelism*, Journal of Functional Programming, **8**(1): 23-60, Jan. 1998.

Efficient Parallelisation of Recursive Problems Using Constructive Recursion*

Magne Haveraaen

Institutt for Informatikk, Universitetet i Bergen, HiB, N-5020 Bergen, Norway
http://www.ii.uib.no/∼magne

Abstract. Many problems from science and engineering may be nicely formulated recursively. As a problem solving technique this is very efficient. Due to the high cost of executing recursively formulated programs, typically an exponential growth in time complexity, this is rarely done in practice. Constructive recursion promises to deliver a space and time optimal compilation of recursive programs. The constructive recursive scheme also straight forwardly generates parallel code on high performance computing architectures.

1 Introduction

Recursive formulation of algorithms has for decades been considered an efficient problem solving method in computer science. The high run-time costs for most recursive programs has been hindering its general applicability. The standard operational interpretation of a recursive program forms a tree of recursive calls spreading out from the node of the initial invocation, i.e., *the root node*, and this is where the result of the computation is gathered. The shape of this tree is defined by the data dependency pattern of the recursive calls. The tree often has many nodes with identical subtrees. The efficiency of the computation, and its parallelisation, crucially depends on identical subtrees being merged into one, thus forming a compact *data dependency graph*. Also, the traversal order of this graph is important: if we trace the data dependency graph from the root, we need to instantiate all the nodes of the data dependency graph before any computation takes place. If we instead proceed in a *data-driven* fashion, starting at the leaves, we will only instantiate nodes when the computation reaches them. A node may be deleted as soon as the execution has passed on the value computed at it to the next node.

Of the three computational models, operational (Turing machines, imperative programs), syntactic (general grammars, term rewriting, λ-calculus and related functional programming), and semantic (μ-recursion), the semantic model readily generalises to recursion on data dependency graphs (from its traditional

* This investigation has been carried out with the support of the European Union, ESPRIT project 21871 SAGA (Scientific Computing and Algebraic Abstractions) and a grant of computing resources from the Norwegian Supercomputer Committee.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 758–761, 2000.
© Springer-Verlag Berlin Heidelberg 2000

recursion on natural numbers). This generalisation is called *constructive recursion* [3, 2]. It includes the definition of the data dependency graph as an explicit component of a recursive program. A compiler may then use this information to create data-driven code distributed on the processors of a parallel machine. We have developed a programming language, Sapphire, embodying these ideas. A Sapphire program consists of three parts: the definition of the data dependency, the definition of the recursive function, and the initiation of the computation.

The next section sketches the principles of constructive recursion and associated compilation technique. Section 3 presents an example with timings. The conclusion gives some references to related work.

2 Constructive Recursion

Somewhat simplistically, a recursively defined function f can be abstracted to the form

$$f_{i_1,\ldots,i_m} = \phi(f_{\delta_1(i_1,\ldots,i_m)}, \ldots, f_{\delta_k(i_1,\ldots,i_m)}),$$

where the tuple $(i_1, \ldots, i_m)$ of variables range over some subset $D \subseteq \mathbb{N}^m$, the *index domain*, of the m-tuple of natural numbers. The functions $\delta_\ell : D \to \mathbb{N}^m$ define the *dependency pattern* or *stencil* of the recursion[1]. The recursion is well defined if the δ_ℓ define a well-funded partial order on the m-tuple of natural numbers $\mathbb{N}^m$, i.e., all paths generated by the δ_ℓ are finite, and there is a set of initial values $\epsilon_{i_1,\ldots,i_m}$ for every index-tuple $(i_1, \ldots, i_m) \in (\mathbb{N}^m \setminus D)$.

To get a constructive recursive definition of f, we also need to know the opposite of the stencil δ_ℓ. The opposite does not mean the inverse of each δ_ℓ. Instead we need a set of functions $\delta'_{\ell'}$ for $\ell' \in \{1, 2, \ldots, k'\}$, such that for every $(i'_1, \ldots, i'_m) \in \mathbb{N}^m$ for which there exists a δ_ℓ and $(i_1, \ldots, i_m)$ such that $\delta_\ell(i_1, \ldots, i_m) = (i'_1, \ldots, i'_m)$, then there exists a $\delta'_{\ell'}$ such that $\delta'_{\ell'}(i'_1, \ldots, i'_m) = (i_1, \ldots, i_m)$. (The number k' of opposite functions may be larger or smaller than k.) Knowing the stencil and its opposite makes us able to trace out the compact data dependency graph from the initial values.

Following [5], parallelising a program is to embed the program's data dependency pattern into the space-time of the target computer's communication structure. This information can be provided by a mapping of the nodes of the dependency graph to space-time coordinates of the target computer. Allowing the programmer to define this mapping, gives explicit control over the parallelisation at a very high abstraction level.

The Sapphire compiler uses this information to generate an imperative program which traces the compact data dependency graph from the initial values. The imperative program generates the nodes of the graph when they are first activated in the computation, and reclaims the nodes as soon as they have been computed and their values passed on in the graph. Parallel code is generated using MPI.

[1] If we limit the form of the δ_ℓ, we get a *recurrence relation*.

3 Example: Heatflow in One Dimension

The one-dimensional heat flow equation, $\alpha^2 \frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}$, describes the heat distribution $u(x,t)$ in a one-dimensional rod along the x-axis at time t. Here α is a physical constant describing the heat transfusion properties of the rod. To turn the equation into a recursion we may use the finite difference method both in time and space, giving

$$u(x,t) = \kappa u(x + \Delta x, t - \Delta t) + (1 - 2\kappa)u(x, t - \Delta t) + \kappa u(x - \Delta x, t - \Delta t),$$

where $\kappa = \alpha^2 \Delta t / \Delta x^2$. This simple program has 3 recursive calls per timestep. Its stencil is given by the δ_ℓ functions.

$$\begin{tikzcd}
 & (x,t) \arrow[dl, "\delta_3"'] \arrow[d, "\delta_2"'] \arrow[dr, "\delta_1"] & \\
(x-\Delta x, t-\Delta t) & (x, t-\Delta t) & (x+\Delta x, t-\Delta t)
\end{tikzcd}$$

As opposites we get the $\delta'_{\ell'}$ functions.

$$\begin{tikzcd}
(x-\Delta x, t+\Delta t) \arrow[dr, "\delta'_1"'] & (x, t+\Delta t) \arrow[d, "\delta'_2"'] & (x+\Delta x, t+\Delta t) \arrow[dl, "\delta'_3"] \\
 & (x,t) &
\end{tikzcd}$$

(By convention we always draw the arrows in the δ_ℓ direction.)

The Sapphire compiler translates this code to non-recursive ANSI C, which traverses the compact graph of the stencils bottom-up. A series of tests were run sequentially on one processor of a SGI Cray Origin 2000 machine. At each timestep the heat distribution for all discrete x-values is computed. The running time of the generated C program clearly grows linearly with the number of timesteps.

No. of Grid Points	No. of Timesteps	Execution Time(sec)	Measured growth	Ideal growth
4000	1000	7.83	1	1
4000	2000	15.64	2.0	2
4000	4000	31.24	4.0	4
4000	8000	62.46	8.0	8

Tests of the MPI version on the SGI Cray Origin 2000 shows good parallel efficiency (1000 timesteps and 80 000 grid points).

No. of Processors	Execution Time(sec)	Measured speed-up	Ideal speed-up	Efficiency Measured/Ideal
2	298	1	1	100%
4	150	2.0	2	100%
8	75	4.0	4	100%
16	38	7.8	8	98%
32	20.5	14.5	16	91%

All tests were run with the machine in ordinary production mode.

4 Conclusion

Recursive programming is a very efficient problem solving technique, but with traditional compiler technology this is often prohibitively expensive to compute. We have suggested supplying the stencil and its opposite as an explicit part of the program. This makes it possible for a compiler to generate code that traces the compact data dependency graph in an efficient data-driven fashion. We can also augment the data dependency graph with information on the distribution of the computation on the processors of a parallel computer. This yields full control over the parallel execution of the code, without the need for the programmer to get involved in the parallel code itself.

Similar approaches have been investigated by other researchers, but in more restricted contexts. In [6] many approaches based on functional programming are reported. Some of these utilise the dependency in order to generate data driven code and parallelise programs, but they do not give the user explicit control over the opposite dependency. This limits the efficiency of the approaches to cases where the dependency may be automatically inverted. In an imperative context Čyras & al. [1] have used dependencies to provide a modular decomposition of the recursive function (with stencils) from code tracing out the graph (the opposite stencils). A detailed comparison with our technique is presented in [2]. See [4] for more detailed examples of the constructive recursion technique.

Acknowledgements Thanks to Steinar Søreide who implemented the Sapphire compiler and did the timings.

References

[1] Vytautas Čyras. Loop synthesis over data structures in program packages. *Computer Programming*, 7:27–50, 1983. (in Russian).

[2] Vytautas Čyras and Magne Haveraaen. Modular programming of recurrencies: a comparison of two approaches. *Informatica*, 6(4):397–444, 1995.

[3] Magne Haveraaen. How to create parallel programs without knowing it. In *Proceedings of the 4th Nordic Workshop on Program Correctness – NWPC4*, number 78 in Reports in Informatics, pages 165–176, P.O.Box 7800, N-5020 Bergen, Norway, April 1993.

[4] Magne Haveraaen and Steinar Søreide. Solving recursive problems in linear time using constructive recursion. In Sverre Storøy, Said Hadjerrouit, et al., editors, *Norsk Informatikk Konferanse – NIK'98*, pages 310–321. Tapir, Trondheim, Norway, 1998.

[5] W.L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32:93–114, 1983.

[6] B.K. Szymanski, editor. *Parallel Functional Languages and Compilers.* ACM Press; New York / Addison Wesley; Reading, Mass., 1991.

Development of Parallel Algorithms in Data Field Haskell*

Jonas Holmerin[1] and Björn Lisper[2]

[1] Department of Numerical Analysis and Computing Science, Royal Institute of Technology, S-100 44 Stockholm, SWEDEN,
joho@nada.kth.se

[2] Dept. of Computer Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, SWEDEN,
bjorn.lisper@mdh.se

Abstract. Data fields provide a flexible and highly general model for indexed collections of data. Data Field Haskell is a dialect of the functional language Haskell which provides an instance of data fields. We describe Data Field Haskell and exemplify how it can be used in the early phase of parallel program design.

1 Introduction

Many computing applications require indexed data structures. The canonical indexed data structure is the array. However, for sparse, distributed applications, other, more dynamic indexed data structures are needed. It is desirable to develop such algorithms on a high level first, in order to get them right, since the low level data representations can be intricate.

Data Field Haskell provides an instance of data fields – a data type for general indexed structures. This Haskell dialect can be used for rapid prototyping of parallel computational algorithms which may involve sparse structures.

Various versions of the data field model have been described elsewere [1, 2, 3, 4]. The contribution of this paper is a description of an implementation and an example of how it can be used in parallel program design.

2 The Data Field Model

Data fields are based on the more abstract model of indexed data structures as functions with finite domain [1, 2]. This model is simple and powerful, but for real implementations explicit information about the function domains is needed. Data fields are thus entities (f, b) where f is a function and and b is a *bound*, a set representation which provides an upper approximation of the domain of the corresponding function. We require that the following operations are defined for bounds:

* This work was supported by The Swedish Research Council for Engineering Sciences (TFR), grant no. 98-653.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 762–766, 2000.
© Springer-Verlag Berlin Heidelberg 2000

- An interpretation of every bound as a set.
- A predicate classifying each bound as either *finite* or *infinite*, depending on whether its set is surely finite or possibly infinite.
- For every bound b defining a finite set, $size(b)$ which yields the size of the set and $enum(b)$ which is a function enumerating its elements.
- Binary operations $\sqcap$, $\sqcup$ on bounds ("intersection", "union").
- The bounds *all* and *nothing* representing the universal and empty set, respectively.

These operations are chosen to support the usually assumed set of collection-oriented operations [7] without revealing the inner structure of the bounds. An important derived operation is *explicit restriction*: $(f, b) \downarrow b' = (f, b' \sqcap b)$. The theory of data fields also defines *φ-abstraction*, a syntax for convenient definition of data fields which parallels λ-abstraction for functions. See [3, 4].

3 Data Field Haskell

Data Field Haskell is a Haskell dialect where the arrays have been replaced by an instance of data fields, a variation of the *sparse/dense arrays* of [3, 4]. Our implementation of Data Field Haskell is based on the NHC compiler [6] for Haskell v. 1.3. The implementation is sequential and we have not implemented any advanced optimizations.

`datafield`	defines data field from function and bound
`!`	data field indexing
`bounds`	the bound of a data field
`outofBounds`	an out-of-bounds error value
`predicate`	forms predicate bound from predicate
`join`, `meet`	"union" and "intersection" of bounds
`<\>`	explicit restriction of data field with bound
`inBounds`	checks if an element belongs to the set defined by a bound
`foldlDf`	folds (reduces) finite data field w.r.t. binary operation

Table 1. Some operations on data fields and bounds.

Data Field Haskell has data types `Datafield a b` for datafields and `Bounds a` for the corresponding bounds. Table 1 lists some functions for data fields and bounds. It has a rich variety of finite and infinite bounds: *dense bounds*, i.e., traditional array bounds, *sparse finite bounds*, which represent general finite sets, *predicate bounds*, which are classified as infinite, `universe` which represents the universal set, and `empty` which represents the empty set. *Product bounds* represent Cartesian products and generalise multidimensional array bounds.

3.1 Forall- and For-Abstraction

Data Field Haskell provides φ-abstraction, with syntax similar to λ-abstraction in Haskell [5]:

$$\texttt{forall}\ apat_1\ \ldots\ apat_n\ \texttt{->}\ exp$$

The semantics of `forall x -> t` is `datafield (\x -> t) b`, where `b` is computed from bounds of data fields in `t` as to give an upper approximation of the domain of `\x -> t`. It can be thought of as an implicitly parallel, functional `forall` statement where first `b` is computed and then, if needed, `\x -> t` is computed for all `x` in `b`.

`for`-abstraction provides a convenient syntax to define a data field by cases over different parts of its domain. The syntax is

$$\texttt{for}\ pat\ \texttt{in}\ \{\ e_1\ \texttt{->}\ e'_1\ \texttt{;}\ \ldots\ \texttt{;}\ e_n\ \texttt{->}\ e'_n\ \}$$

where the e_i are bounds and the e'_i are data field values. The semantics is a data field whose bound is restricted by the union of $e_1, \ldots, e_n$ and whose value for each `x` is e'_i `! x`, for the lowest i such that `x` belongs to e_i.

4 An Example

We exemplify the use of Data Field Haskell with simulation of a system of particles over a range of time. Each particle i has a state $(\bar{r}_i, \bar{v}_i)$ with a position $\bar{r}_i$ and a velocity $\bar{v}_i$. The state transition function (1) gives the new state after time Δt. These equations are parameterized w.r.t. f_i, which yields the acceleration a_i for each particle i from the spatial distribution of the particles. They are more or less directly expressed[1] in Data Field Haskell in Fig. 1. The result is an executable, dimension-polymorphic specification of particle simulation with time step `dt`.

$$\begin{aligned} (\bar{r}_i, \bar{v}_i) &= (\bar{r}_i + \bar{v}_i \cdot \Delta t, \bar{v}_i + \bar{a}_i \cdot \Delta t), \quad i \in \mathit{Particles} \\ \bar{a}_i &= f_i(\langle\, r_j \mid j \in \mathit{Particles}\,\rangle), \quad i \in \mathit{Particles} \end{aligned} \tag{1}$$

This specification can be manually refined into a more explicitly parallel algorithm for particle simulation by *distributing* the particle state onto a set of processors, and *localizing* the parallel algorithm by neglecting long-range interactions. The resulting parallel algorithm has a distributed state which consists of a predicate, a sparse particle datafield, and a *neighbourhood bound* for each processor. A processor "owns" each particle in the area defined by its predicate. After each iteration every processor tests which particles it now owns. This test is localized to particles originating from the neighbours only. This approximation is correct only if long-range interactions can be ignored, for instance if we

[1] The explicit restriction in `p_newstate` is necessitated by a flaw in the current derivation of bounds for `forall`-abstraction. We expect to rectify it in later releases of Data Field Haskell.

simulate molecules which interact through collisions only. The set of neighbour processors is described, for each processor, by its neighbourhood bound. Fig. 1 also gives the data type and transition function `d_newstate` for the distributed particle simulation. `d_newstate` first computes, for each processor, the union of the particles over the neighbours, then a local state transition for these, and finally it "masks out" the particles which now belong to it. (Since data fields are not sets we must instead of set union use "`prunion`", which gives priority to its first argument for indices where both arguments are defined.) `d_newstate` does not modify the predicate field, but this is possible and could be used to model adaptive methods where areas for different processors are changed to balance the load.

```
data Pstate = PS (Datafield Int Float) (Datafield Int Float)
pos (PS r _) = r
vel (PS _ v) = v

p_newstate dt f s =
  forall i ->
    let ai = f i s in
      PS ((forall k-> ((pos (s!i))!k) + ((vel (s!i))!k)*dt)
            <\> bounds (pos (s!i)))
         (forall k-> ((vel (s!i))!k) + (ai!k)*dt)


data Dstate proc_id part_id = DS ((Datafield Int Float) -> Bool)
                                 (Datafield part_id Pstate)
                                 (Bounds proc_id)
pred  (DS p _ _) = p
state (DS _ s _) = s
neigh (DS _ _ b) = b

prunion d1 d2 =
  forall x -> if (inBounds x (bounds d1)) then d1!x else d2!x
prUnion = foldlDf prunion (datafield (\x -> outofBounds) empty)

d_newstate dt f dstate =
  forall p ->
    DS (pred (dstate!p))
       (let {
          neighstate = prUnion
                       (for pn in neigh (dstate!p) -> state (dstate!pn));
          newstate   = (p_newstate dt f neighstate) }
        in newstate <\> predicate(\j -> pred (dstate!p)(pos(newstate!j))))
       (neigh (dstate!p))
```

Fig. 1. Data Field Haskell solutions for the simulation problem: executable specification, and parallelised version.

5 Conclusions

We have presented Data Field Haskell, a Haskell dialect with data fields which supports a highly flexible form of collection-oriented programming. A possible application is for rapid prototyping in the early specification phase of parallel algorithms. We exemplified with the specification and initial development of a simple parallel particle simulation algorithm. The resulting algorithm specification is dimension-polymorphic and easily adaptable to regular and irregular grids, static or dynamic mappings, and different target architectures.

References

[1] P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 210–222. ACM Press, June 1993.

[2] B. Lisper. Data parallelism and functional programming. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Vol. 1132 of *Lecture Notes in Comput. Sci.*, pages 220–251, Les Ménuires, France, Mar. 1996. Springer-Verlag.

[3] B. Lisper. Data fields. In *Proc. Workshop on Generic Programming*, Marstrand, Sweden, June 1998. `http://wsinwp01.win.tue.nl:1234/WGPProceedings/`.

[4] B. Lisper and P. Hammarlund. The data field model. Technical Report TRITA-IT R 99:02, Dept. of Teleinformatics, KTH, Stockholm, Mar. 1999. `ftp://ftp.it.kth.se/Reports/TELEINFORMATICS/TRITA-IT-9902.ps.gz`.

[5] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, E. Meijer, S. L. Peyton Jones, A. Reid, and P. Wadler. Report on the programming language Haskell: A non-strict purely functional language, version 1.4, Apr. 1997. `http://www.haskell.org/definition/`.

[6] N. Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1995.

[7] J. M. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, Apr. 1991.

The ParCeL-2 Programming Language*

Paul-Jean Cagnard

Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland
cagnard@di.epfl.ch

Abstract. A semi-synchronous parallel programming language and the corresponding computing model are presented. The language is primarily designed for massively parallel fine grained applications such as cellular automata, finite element methods, partial differential equations or systolic algorithms. In this language it is assumed that the number of parallel processes in a program is much larger than the number of processors of the machine on which it is to be run. The computational model and the communication model of the language are presented. Finally, the language syntax and some performance measurements are presented.

1 Introduction

Many parallel programming models have been proposed in the past years as alternatives to both message passing and data parallel programming. Among the most prolific class of alternate models there is the family of languages based on concurrent objects or actors [1, 2].

In actor- or object-based models, parallelism of execution is expressed explicitly. These models allow to express the distribution of data explicitly and in a natural way. This opens interesting opportunities for automated data- and process-mapping.

Nearly all models based on concurrent objects or actors use asynchronous execution models, i.e., execution schemes where computation and message sending are not inherently automatically globally synchronised. This entails that communication is not easy to implement efficiently and concurrent object systems are often restricted to coarse grain parallelism for performance reasons.

In the following a parallel programming language whose computation model can be seen as an extension of the BSP [3] model and other synchronous object-oriented models is presented.

2 The ParCeL-2 Programming Model

A ParCeL-2, program is typically composed of a large number of fine grained processes executing concurrently. Each process execution consists of a sequence of supersteps. A superstep is composed of two phases: a computation phase and a communication phase. Supersteps are separated by synchronisation barriers.

* Work supported by the Swiss National Science Foundation under grant 21-49519.96.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 767–770, 2000.
© Springer-Verlag Berlin Heidelberg 2000

During the computation phase of a superstep, a process executes computations which only manipulate data local to this process. A process can send data to other processes in the course of a computation phase. The effective transmission of data between processes happens at the end of each superstep. This means that no data are exchanged during computation phases. It can be observed that this computing model is an intrinsically distributed memory MIMD model.

Aggregates of processes can be constructed in order to build abstract processes whose behaviour is more complex than an elementary process but the rest of the program does not need to know how this behaviour is implemented. Aggregates of processes offer means of creating arbitrarily complex process topologies and hierarchical parallel program designs.

In ParCeL-2, unlike in BSP, every process, or cell, can execute supersteps with a slower frequency than that of the execution environment's global clock. This is why ParCeL-2 is called semi-synchronous. This is useful since complex cells containing other cells may need more time to produce a meaningful result than an elementary cell. This is a way of saying that if a cell has a synchronisation period n, data will only send along or read from its channels every n elementary supersteps.

In ParCeL-2, processes can only communicate through predeclared channels. Channels have four main properties. They are typed: a channel, like a local variable, is of a certain type, and only data of this type can be sent through this channel. They are directed: channels are unidirectional. If bidirectional communications are needed, two channels of opposite directions must be created between processes. They have a period which is the same as the period of the cell at the start of the channel. They are static: this means that processes connected at the other end of a channel can't change during execution, and that channels can't be created nor destroyed at runtime. Processes are also static, they can't be created nor destroyed at runtime.

In the case of complex cells, there is a special type of connection allowed, called a *shunt* which is useful when the complex cell is only a structural cell, i.e. has no body, or when some of the internal cells need to pass data to the exterior of the complex directly.

Channels do not necessarily connect only one process to another single process, they may connect many processes together. Thus three kinds of channels may be declared in a program: 1-1 channels, 1-n channels, and n-1 channels in which case an operator can be attached to the channel to reduce n values into a single one.

One of the goals of ParCeL-2 and one of its main features is to provide a topology description language in order to be able to describe easily, elegantly and concisely complex topologies. Such topology description languages have been developed, see [4, 5], but neither of them permits to construct topologies with all the features of ParCeL-2 channels.

3 The ParCeL-2 Syntax

The declaration of a type of cells consists of three main parts: an interface, a body or *behaviour*, and a *topology*.

3.1 Interface Declarations

In the interface part of a cell type, one can put type declarations, constant declarations and channel declarations. Type and constant declarations are written according to the syntax of the language chosen for the body of the cell type.

An **out** channel is always declared the same way, whether it will be connected as a 1-1, 1-n or n-1 channel:

```
OUT <data_type> <channel_name>
```

An **in** channel can be declared in three ways according to the chosen kind of utilisation, respectively 1-1 or 1-n, n-1 mailbox , and n-1 with reduction.

```
IN <data_type> <channel_name> [= <init_value>]
IN(mbox) <data_type> <channel_name> [= <init_value>]
IN(reduce, <operator>) <data_type> <channel_name> [= <init_value>]
```

The `import <cell_type>` statement permits to reuse cell code.

3.2 Body Declarations

In the body part of a cell type, anything that can be written in the chosen programming language can be written. With the exception that some keywords are reserved by ParCeL-2. That code will be executed at each superstep.

3.3 Topology Declarations

This is in that part of a cell type that internal cells are instantiated and connected according to the topology desired by the programmer. This is what differentiates elementary cells from complex cells. Complex cells embed several instances of other types of cells, which are called *internal cells*, and the topology describes how these instances are connected together and with the embedding cell. The topology description language is currently still under development. Work is underway to provide construction operators like the cartesian product of two graphs and the joining of two sub-topologies, like in Candela [6]. The language will be shown through two examples which illustrate a highly regular and a highly irregular topology. The cell types used are supposed to be declared somewhere else.

```
ring(n) of TA
{
  aux = array(1..n) of TA;
  for i in 1..n
```

```
    connect(aux[i].out1,aux[(i mod n) + 1].in1);
    connect(aux[i].out2,aux[(i mod n) + 1].in2);
    connect(aux[(i mod n) + 1].out3, aux[i].in3);
  end for;
}
ring(3) of TA;

A = new T1; B = new T2; C = new T3;
connect(A.out1,B.in1); connect(B.out1,C.in1); connect(C.out1,A.in1);
connect(A.g,g); connect(h,B.h); shunt(C.e,c); shunt(d,e.f);
```

4 Conclusion and Future Work

A programming language particularly well suited to the design of massively parallel fine grained applications has been presented. Complex computations using cellular automata, partial differential equations, finite element methods, systolic arrays or even static neural networks can benefit much from such a language, notably in portability and readability.

A first version of the runtime environment has been implemented and used to run a program implementing the game of life with a grid of 1000×1000 cells on a Fast Ethernet network of 333 MHz Sun Ultra 10 workstations using MPI as a communication library. For 50 iterations, the best sequential program took 3.3 s. The parallel version was slower, but scaled quite well, from 22.5 s to 3.89 s, using 1 to 16 processors, which is very satisfying considering that our runtime environment is not optimized at all yet. Now its behaviour must be studied with real world applications like wave propagation [7] for example.

Work on a full compiler remains to be done. It exists only in parts at the moment.

References

[1] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI-73*, 1973.

[2] G. Agha. *ACTORS, a model of concurrent computation in distributed systems.* MIT Press, 1986.

[3] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[4] Alexey Lastovetsky. mpC - a multi-paradigm programming language for massively parallel com puters. In *ACM SIGPLAN Notices*, volume 31(2), pages 13–20. February 1996.

[5] András Varga. Parametrized topologies for simulation programs. In *Western Multiconference on Simulation (WMC'98) / Communication Network s and Distributed Systems (CNDS'98)*, San Diego, CA, January 1998.

[6] Herbert Kuchen and Holger Stoltze. Candela – a topology description language. *Journal of Computers and Artificial Intelligence*, 13(6):557–676, 1994.

[7] F. Guidec, P. Calégari, and P. Kuonen. Parallel irregular software for wave propagation simulation. *Future Generation Computer Systems (FGCS), N.H. Elsevier*, 13(4-5):279–289, March 1998. ISSN 0167-739X.

Topic 11
Numerical Algorithms for Linear and Nonlinear Algebra

Ulrich Rüde and Hans-Joachim Bungartz

Topic Chairmen

Since the early days of supercomputing, numerical algorithms have been the application with the highest demand for computing power anywhere. Many of today's fastest computers in the world, as given in the TOP 500 list[1], are mostly used for the solution of huge systems of equations as they arise in the simulation of complex large scale problems in engineering and science.

In contrast to the situation a decade ago when vector processors still dominated the supercomputer market, now, all high-performance computers are parallel architectures. Furthermore, the number of processors in current state-of-the-art supercomputer systems ranges from 100 to 10000, corresponding to moderate to massive parallelism. The most powerful of these high-end computers currently perform beyond one Teraflop, and expected future Petaflop systems will even have to further increase parallelism, thus increasing the demand for efficient parallel numerical algorithms, too.

The answer to the question whether it will be possible to exploit the full potential of such massively parallel systems will, therefore, primarily depend on future progress in the development of parallel algorithms for large scale algebraic systems and related numerical problems. This crucial importance of parallel numerical algorithms certainly justifies devoting a special workshop to this topic at a conference on parallel systems and algorithms like Euro-Par, in addition to the discussion of special aspects of high-performance computing in Topic 7 (*Applications on High-Performance Computers*).

Recently, as numerical simulation has become more and more a standard technology in academic and industrial research and development, parallel numerical algorithms have also gained in importance outside the field of classical supercomputers. Clusters of networked personal computers or workstations, as they are in the focus of Topic 18 (*Cluster Computing*), do no longer suffer from the image of being the poor man's supercomputer, only. Among other arguments, the better availability as well as an often more favourable cost effectiveness of the cluster approach have obviously helped to raise industry's interest in parallel or distributed solutions. Hence, today, we face an increasing demand for parallel engineering applications on clustered systems.

Overall, fifteen papers were submitted to our Topic. With authors from Croatia, Germany, Greece, Italy, Russia, Spain, and the United Kingdom, Europe is the dominant male (as expected at Euro-Par), but four papers are authored

[1] www.top500.org

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 771–773, 2000.
© Springer-Verlag Berlin Heidelberg 2000

or co-authored by scientists from the United States and from China. Out of these fifteen submissions, nine were accepted (six as regular papers and three as research notes).

Both devising new numerical algorithms and adapting existing ones to the current parallel systems are vigorously flourishing and still developing fields of research. Hence, it is no surprise that the ten research articles presented in this section cover a wide range of topics arising in the various subdomains of parallel numerical algorithms. At the conference, the presentations were arranged into three sessions on *Numerical Linear Algebra*, *Algorithms for Partial Differential Equations*, and *Ordinary Differential Equations and other Topics*. This structure also reflects in the following part of the conference's proceedings.

Many if not most large scale algebraic systems arise out of the discretization of (partial) differential equations. Hence, the resulting algebraic systems are sparse. Exploiting the sparsity structure is the key to an efficient solution. However, this requires approaches tailored to the respective situation. Here, many of the leading algorithms are iterative schemes. Thus, numerical methods for partial differential equations and numerical linear algebra are closely related, which may let appear the subdivision of the talks in the first two sections somewhat arbitrary.

In the *Numerical Linear Algebra* section, David S. Wise discusses the Ahnentafel indexing and Morton-ordered arrays, a technique for improving the locality of multi-dimensional arrays (like sparse sets of points or sparse matrices) in memory. This is of importance for both improving the cache performance on a single processor and for partitioning matrices across processors without entailing large communication during matrix operations. Peter Gottschling and Wolfgang E. Nagel deal with practical experience with the cascadic conjugate gradient method, an algorithm with an asymptotic performance similar to multigrid methods, but with a simpler access to parallelization.

The contributions in the session on *Algorithms for Partial Differential Equations* deal with the fast solution of large linear systems and with grid generation. For an efficient parallelization of PDE solvers, already the grid generation and partitioning must be performed in parallel. The presentation given by Jörn Behrens and Jens Zimmermann is concerned with the parallel generation of unstructured grids based on the idea of space-filling curves, recently used for parallelization purposes. The talk of Michael Bader and Christoph Zenger deals with a robust multigrid solver for systems stemming from convection-diffusion equations, based on the idea of the iterative nested dissection. The third talk, given by Marcus Mohr, reports on a variant of multigrid where special features of the algorithm are exploited to reduce the amount of communication between subdomains.

The final session gathers various topics. Peter Benner, Rafael Mayo, Enrique S. Quintana-Orti, and Vicente Hernandez deal with a cluster solver for the discrete-time periodic Riccati equations. Furthermore, there are two contributions on optimization problems. In the presentation by Torsten Butz, Oskar von Stryk, Thieß-Magnus Wolter, and Cornelius Chucholowski, the optimization

arises out of a parameter estimation problem in vehicle dynamics as a genuinely industrial application on a cluster of PCs. The talk by Marco D'Apuzzo, Marina Marino, Panos M. Pardalos, and Gerardo Toraldo deals with box-constrained quadratic programming. Finally, Christos Kaklamanis, Charalampos Konstantopoulos, and Andreas Svolos discuss a parallel implementation of a sliding-window compression algorithm for hypercube processor topologies.

Altogether, the contributions to Topic 11 at the millennium Euro-Par in Munich show once more the great variety of interesting, challenging, and important issues in the field of parallel numerical algorithms. Thus, we are already looking forward to the results presented at next year's Euro-Par conference.

AHNENTAFEL INDEXING INTO MORTON-ORDERED ARRAYS, or MATRIX LOCALITY FOR FREE*

David S. Wise**

Indiana University

Abstract. Definitions for the uniform representation of d-dimensional matrices serially in Morton-order (or Z-order) support both their use with cartesian indices, and their divide-and-conquer manipulation as quaternary trees. In the latter case, d-dimensional arrays are accessed as 2^d-ary trees. This data structure is important because, at once, it relaxes serious problems of locality and latency, and the tree helps schedule multiprocessing. It enables algorithms that avoid cache misses and page faults at all levels in hierarchical memory, independently of a specific runtime environment.

This paper gathers the properties of Morton order and its mappings to other indexings, and outlines for compiler support of it. Statistics elsewhere show that the new ordering and block algorithms achieve high flop rates and, indirectly, parallelism without any low-level tuning.

CCS Categories and subject descriptors: E.1 [Data Structures]: Arrays; D.3.2 [Programming Languages]: Language Classifications–concurrent, distributed and parallel languages; applicative languages; D.4.2 [Operating Systems]: Storage management–storage hierarchies; E.2 [Data Storage Representations]: contiguous representations.
General Term: Design.
Additional Key Words and Phrases: caching, paging, compilers, quadtree matrices.

1 INTRODUCTION

Maybe we've not been representing them efficiently for some time. Matrix problems have been fodder for higher-level languages from the beginning [1], and row- or column-major representations for matrices are universally assumed. Both use the same space, both still survive.

But maybe both are archaic perspectives on matrix structure, which might best be represented using a third convention. Architecture has developed quite a bit since we had to pack scalars into small memory. With hierarchical—rather than flat—memory, only the faster memory is precious; with distributed processing instructions on local registers are far faster than those touching remote memory. And, of course, multiprocessing on many cheap processors demands less crosstalk than code for single threading on uniprocessors with unshared

* Copyright ©2000 to Springer-Verlag, with rights reserved for anyone to make digital or hard copies of part or all of this work for personal or classroom use, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full Springer citation on the first page. Rights are similarly reserved for any library to share a hard copy through interlibrary loan.

** Supported, in part, by the National Science Foundation under grants numbered CDA93–03189 and CCR-9711269. Author's address: Computer Science Dept, 215 Lindley Hall, Bloomington, IN 47405–7104, USA. dswise@cs.indiana.edu

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 774-783, 2000.
© Springer-Verlag Berlin Heidelberg 2000

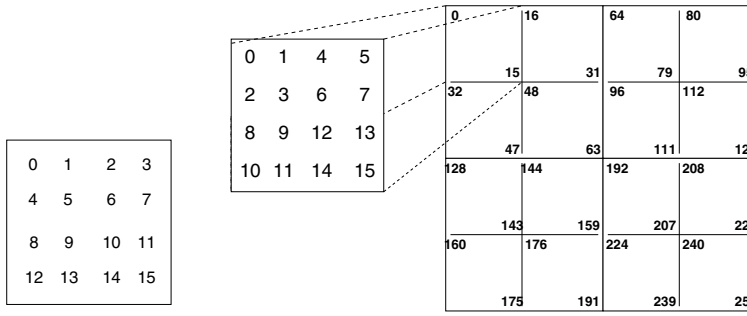


Figure 1. Row-major indexing of a 4×4 matrix, and analogous Morton indexing.

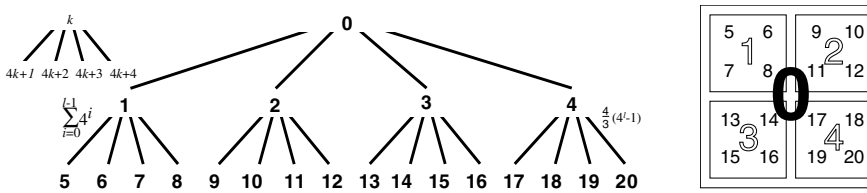


Figure 2. Level-order indexing of the order-4 quadtree and its submatrices.

memory. Address space, itself, has grown so big that chunks of it are extremely cheap, when mapped to virtual memory and never touched. (Intel's IA-64 processor has three levels of cache, two on board.) But fast cache, local to each processor, remains dear, and, perhaps, row-major and column-major are exactly wrong for it.

This paper enhances Morton-order (also called Z-order) storage for matrices. Consistent with the conventional sequential storage of vectors, it also provides for the usual cartesian indexing into matrices (row, column indices). It extends to higher dimensional matrices. That is, we can provide cartesian indexing for "dusty decks" while we write new divide-and-conquer and recursive-descent codes for block algorithms on the same structures. HASKELL and ML could share arrays with FORTRAN and C. Thus, parallel processing becomes accessible at a very high level in decomposing a problem. Best of all, the content of any blocks is addressed sequentially and blocks' sizes vary naturally (they undulate) to fit the chunks transferred between levels of the memory hierarchy.

2 BASIC DEFINITIONS

Morton presented his ordering in 1966 to index frames in a geodetic data base [11]. He defines the indexing of the "units" in a two-dimensional array much as in Figure 1, and he points out the truncated indices available for enveloping blocks (subtrees), similar to Figure 3. Finally, he points out the conversion to and from cartesian indexing available through bit interleaving.

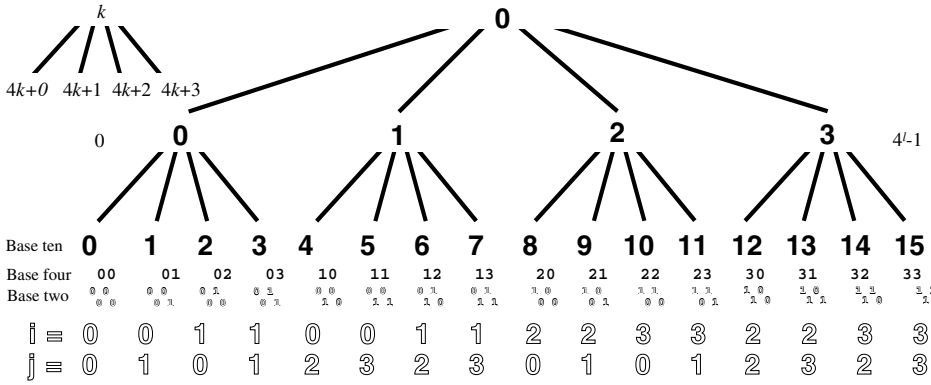


Figure 3. Morton indexing of the order-4 quadtree.

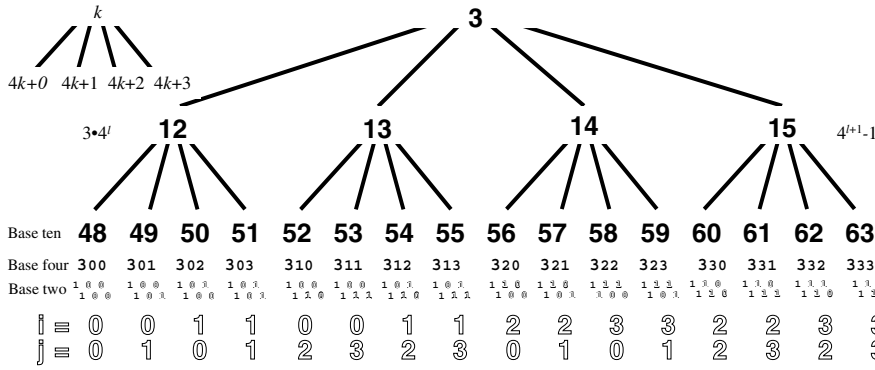


Figure 4. Ahnentafel indexing of the order-4 quadtree.

The definitions later focus on two-dimensional arrays: matrices. The early ones are general: a d-dimensional array is decomposed as a 2^d-ary tree.

2.1 ARRAYS

Definition 1 *In the following* $\mathrm{m} = 2^d$ *is the degree of the tree appropriate to the dimension* d.
If the maximal order in any dimension is n then the tree has maximal level $\lceil \lg n \rceil$. Use Figures 2, 3, and 4 in reading the following definitions.
Definition 2 *A complete array has* level-order *index* 0. *A subarray (block) at level-order index* i *is either a scalar, or it is composed of* m *subarrays, with level-order indices* $mi+1, mi+2, \ldots, mi+m$.
Definition 3 *The root of an array has* Morton-order *index* 0. *A subarray (block) at Morton-order index* i *is either a unit (scalar), or it is composed of* m *subarrays, with indices* $mi+0, mi+1, \ldots, mi+(m-1)$ *at the next level.*

Theorem 1. *The difference between the level-order index of a block at level l in an array and its Morton-order index is* $(m^l - 1)/(m - 1)$.

The difference is the number of nonterminal nodes above level l. Since each level is indexed by its zero-based scheme, it is necessary also to know the level and a Morton index, to identify a specific node.

Ahnentafel indices are immensely useful for identifying blocks at all levels [17]. Algorithms that use recursive-descent (divide-and-conquer) to descend to a block of arbitrary size, or to return the index of a selected block, need only this single index to identify any subtree. Treat them only as identification numbers because there are gaps in the sequence between levels. Conversions among Ahnentafel indices, cartesian indices, Morton order, and level order are easy.

Ahnentafel indices come to us from genealogists who invented them for encoding one's pedigree as a binary, family tree. This generalization to m-ary trees is new.

Definition 4 [4] *A complete array has* Ahnentafel *index* $m - 1$. *A subarray (block) at Ahnentafel index* a *is either a scalar, or it is composed of* m *subarrays, with indices* $ma + 0, ma + 1, \ldots, ma + (m - 1)$.

Theorem 2. *The nodes at level l have Ahnentafel indices from* $(m-1)m^l$ *to* $m^{l+1} - 1$. *The gap in indices between level* $l - 1$ *and level l is* $m^l(m - 2) + 1$.

Theorem 3. *The level of a node with Ahentafel index a is* $l = \lfloor \log_m a \rfloor = \lfloor \log_m \frac{a}{m-1} \rfloor$. *The difference between the Ahnentafel index and the level-order index is* $(m^{l+1}(m - 2) + 1)/(m - 1)$. *The difference between the Ahnentafel index and the Morton-order index is* $(m - 1)m^l$.

The gaps between levels in Ahnentafel indexing of quadtrees do not arise in binary trees. The strong similarity between level-order and Ahnentafel indexing in this common case perhaps explains why the latter has been often overlooked. For instance, Knuth's level-order indexing, based at one, is off-by-one relative to Definition 2 [10, p. 401], but coincides with Ahnentafel indexing on binary trees.

2.2 MATRICES

Hereafter, we assume $d = 2$ for matrices; so $m = 4$. In all the figures, the cartesian indices of the leaves appear in outlined font below the tree.

Corollary 1. *The difference between level-order index of a matrix block at level l and its Morton-order index is* $(4^l - 1)/3$.

Corollary 2. *The gap in Ahnentafel indices between level* $l - 1$ *and l is* $2^{2l+1} + 1$.

Corollary 3. *The difference between the Ahnentafel index of a submatrix at level l and its level-order index is* $(2^{2l+3} + 1)/3$.
The difference between its Ahnentafel and its Morton-order index is $3 \cdot 4^l$.

Definition 5 *Let* w *be the number of bits in a (short) word.*
Each $\mathrm{q_k}$ *is a modulo-4 digit (or* quat*).*
Each q_k *is alternatively expressed as* $q_k = 2i_k + j_k$ *where* $\mathrm{i_k}$ *and* $\mathrm{j_k}$ *are* bits.

Cartesian indices have w bits; Morton indices (and, later, dilated integers) have $2w$ bits.

Theorem 4. [11] *The Morton index* $\sum_{k=0}^{w-1} q_k 4^k = 2\sum_{k=0}^{w-1} i_k 4^k + \sum_{k=0}^{w-1} j_k 4^k$ *corresponds to the cartesian indices: row* $\sum_{k=0}^{w-1} i_k 2^k$ *and column* $\sum_{k=0}^{w-1} j_k 2^k$

The proof is by simple induction on w; doubling the order of a matrix introduces two high-order bits. The quats, read in order of descending subscripts, select a path from the root to the node, as in Figure 4. For example, if $i = 4 = 10_4$ and $j = 8 = 20_4$ in Figure 1, the Morton index is $200_4 + 1000_4 = 3000_4 = 96_{10}$.

Corollary 4. *Let* $l = \lfloor \log_4 a \rfloor$. *The Ahnentafel index,* $a = \sum_{k=0}^{l} q_k 4^k$ *corresponds to the Morton index* $\sum_{k=0}^{l-1} q_k 4^k$.

The bits, $\{i_k\}$, are the odd-numbered bits in the Morton index, and the $\{j_k\}$ are just the even-numbered bits; and, excluding Corollary 3's two high-order **1** bits, coincident with those in an Ahnentafel index. This is Morton's bit interleaving of cartesian indices.

Code to convert from cartesian indices to a Morton index by shuffling bits—or the inverse conversion that deals out the bits—can be slow. Fortunately, as the next section shows, most conversions can be elided.

Definition 6 *The integer* $\overrightarrow{\mathrm{b}} = \sum_{k=0}^{w-1} 4^k$, *here labeled* `evenBits` *in C, and is the constant* `0x55555555`*). Similarly,* $\overleftarrow{\mathrm{b}} = 2\overrightarrow{b}$ *is called* `oddBits`, `0xaaaaaaaa`*).*

In C code $\overrightarrow{b}$ is a very important constant available independently of w as `((unsigned int)-1)/3)`. Masking a Morton index with $\overrightarrow{b}$ or $\overleftarrow{b}$ extracts the bits of the column and row cartesian indices. Morton describes how to use these to obtain indices of neighbors. Their easy identification makes the indexing attractive for graphics in two dimensions and for spatial data bases in three.

It is remarkable how often these basic properties of Morton ordering have been reintroduced in different contexts [3, 7, 9, 12, 16]. Samet gives an excellent history [13].

When first introduced, it might appear that Morton indexing is poor for large arrays that are not square or whose size is not a power of two, because in those cases its gaps seem to waste space. With the valid elements justified to the north and west, the "wasted" space lands in the south and east. It can be viewed as padding, perhaps all zero elements. However, the perceived waste is merely *address* space. In hierarchical memory only its margin will ever be swapped into cache. That is, the "wastage" exists only within the logical/physical addressing of swapping disk; little valuable, fast memory is lost.

With these orders defined and the various index translations among them understood, we can summarize the programmer's view of Morton order:

- The elements of a matrix (an array) are mapped onto memory in Morton order.
- Row and column traversals can still be handled. See the next section.
- If necessary with cartesian indexing, restrict blocking to submatrices implicit in the quadtree (those with Ahnentafel indices.)
- If data is associated also with nonterminal blocks, store it in level order.
- Use Ahnentafel indices to control recursive-descent algorithms [17].
- Ahnentafel indices are monotonic across any row or column, so bounds checking remains available via masking with `evenBits` or `oddBits`.

An early context for implementation of this design is HASKELL whose higher-dimension array aggregates (also, array comprehensions) do not imply any particular internal representation; that is, no code depends on a particular ordering. Moreover, asynchronous, parallel algorithms are implict in that style. Not accidentally, implementations of HASKELL and other functional languages do a good job implementing recursion in preference to iteration. Nevertheless, the algebra of indices implemented in the strength reduction and loop unrolling of FORTRAN and C compilers does not yet have an analog under recursion. Morton order, with its own algebra of indexing and recursion unfolding, offers this leverage and opens access to new scientific algorithms with functional style.

3 CARTESIAN INDEXING AND MORTON ORDERING

The following techniques for cartesian indexing of Morton-order arrays seem to be hardly known, a fact that is unfortunate because they make the structure useful for blocking matrices even if only used with cartesian indexing. In particular, this section newly shows how to compile row and column traversals (of blocks at any level of the tree) with the reductions in operator strength associated with optimizing compilers.

3.1 DILATED INTEGERS

The algebra of dilated integers is surprisingly old. Tocher outlined it in 1954 and under similar constraints to those again motivating us: non-flat memory with access time dependent on locality, and nearby information more rapidly accessible [15, p. 53–55]. But how the size of the memory has changed! Tocher needed fast access into a $32 \times 32 \times 32$ boolean array stored on a drum (4kB!).

Schrack shows how to effect efficient cartesian indexing from row i and column j indices into Morton-order matrices [14]. The trick is to represent i and j as dilated integers, with information stored only in every other bit.

Definition 7 *The* even-dilated representation *of* $j = \sum_{k=0}^{w-1} j_k 2^k$ *is* $\sum_{k=0}^{w-1} j_k 4^k$, *denoted* $\overrightarrow{j}$. *The* odd-dilated representation *of* $i = \sum_{k=0}^{w-1} i_k 2^k$ *is* $2\overrightarrow{i}$, *denoted* $\overleftarrow{i}$.

The arrows suggest the justification of the meaningful bits in either dilated representation. For example, the right arrow suggests rightmost Bit 0 and even kin.

Theorem 5. *A matrix of* m *rows and* n *columns should be allocated as a sequential block of* $\overleftarrow{m-1} + \overrightarrow{n-1} + 1$ *scalar addresses.*

This value is, of course, the Morton index of the southeast-most element of the matrix, plus one for (the northwest-most) one whose Morton index is 0. Not all of that sequence need be active; undefined data at the idle addresses will remain resident in the lowest level of the memory hierarchy. Only data in the active addresses will ever migrate to cache.

Theorem 6. [14] *If "≡" is read as semantic equivalence and "=" denotes equality on integer representations, then for unsigned integers*

$$(\overrightarrow{i} = \overrightarrow{j}) \equiv (i = j) \equiv (\overleftarrow{i} = \overleftarrow{j});$$

$$(\overrightarrow{i} < \overrightarrow{j}) \equiv (i < j) \equiv (\overleftarrow{i} < \overleftarrow{j}).$$

So comparison of dilated integers is effected by the same processor commands as those for ordinary integers.

Definition 8 *The following theorems apply to w-bit 2's-complement integers. The infix operator $\wedge$ indicates bitwise conjunction; $\vee$ denotes bitwise disjunction.*

Instead of representing i and j internally, represent them as $\overleftarrow{i}$ and $\overrightarrow{j}$.

Theorem 7. *The Morton index for the $\langle i, j\rangle^{\text{th}}$ element of a matrix is $\overleftarrow{i} \vee \overrightarrow{j}$, equivalently $\overleftarrow{i} + \overrightarrow{j}$.*

Often (*normalized* dilated integers [14]) addition will be used instead of disjunction to associate at compile time with an adjacent addition.

Addition and subtraction of dilated integers can be performed with a couple of minor instructions.

Definition 9 *Addition ($\overrightarrow{+}, \overleftarrow{+}$) and subtraction ($\overrightarrow{-}, \overleftarrow{-}$) of even- and odd-dilated integers:*

$$\overrightarrow{j} \overrightarrow{-} \overrightarrow{n} = \overrightarrow{j - n}; \qquad \overleftarrow{i} \overleftarrow{-} \overleftarrow{n} = \overleftarrow{i - n}.$$

$$\overrightarrow{j} \overrightarrow{+} \overrightarrow{n} = \overrightarrow{j + n}; \qquad \overleftarrow{i} \overleftarrow{+} \overleftarrow{n} = \overleftarrow{i + n}.$$

Theorem 8. *Subtraction, addition, constant addition, and shifts on dilated integers:*

$$\overrightarrow{j} \overrightarrow{-} \overrightarrow{n} = (\overrightarrow{j} - \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad \overleftarrow{i} \overleftarrow{-} \overleftarrow{n} = (\overleftarrow{i} - \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [14]$$

$$\overrightarrow{j} \overrightarrow{+} \overrightarrow{n} = (\overrightarrow{j} + \overleftarrow{b} + \overrightarrow{n}) \wedge \overrightarrow{b}; \qquad \overleftarrow{i} \overleftarrow{+} \overleftarrow{n} = (\overleftarrow{i} + \overrightarrow{b} + \overleftarrow{n}) \wedge \overleftarrow{b}; \qquad [14]$$

$$\overrightarrow{i} \overrightarrow{+} \overrightarrow{c} = \overrightarrow{i} \overrightarrow{-} \overrightarrow{(-c)}; \qquad \overleftarrow{i} \overleftarrow{+} \overleftarrow{c} = \overleftarrow{i} \overleftarrow{-} \overleftarrow{(-c)};$$

$$\overrightarrow{b} = \overrightarrow{(-1)}; \qquad \overleftarrow{b} = \overleftarrow{-1};$$

$$\overrightarrow{i<<k} = \overrightarrow{i} << (2k); \qquad \overleftarrow{i<<k} = \overleftarrow{i} << (2k);$$

$$\overrightarrow{i>>k} = \overrightarrow{i} >> (2k); \qquad \overleftarrow{i>>k} = \overleftarrow{i} >> (2k).$$

Theorem 8 and 6 suggest that the C loop

```
for (int i=0; i<n; i++){...}
```

be compiled to `int nn=`$\overleftarrow{n}$ and

```
for (int ii=0; ii<nn; ii=(ii-oddBits)&oddBits ){...}
```

So, if i and j are not represented literally as integers, but translated by the compiler to their images, $\overleftarrow{i}$ and $\overrightarrow{j}$, the resulting object code can be just a simple homomorphic image of what the programmer expected. Code like the following source might be demanded from the programmer, but it would be better introduced via a transformation by the helpful compiler:

```
#define evenBits ((unsigned int) -1)/3)
#define oddBits  (evenBits <<1)
#define  oddIncrement(i)  (i= ((i -  oddBits) &  oddBits))
#define evenIncrement(j)  (j= ((j - evenBits) & evenBits))
...
   for     (i = 0; i< rowCountOdd ;  oddIncrement(i))
     for   (j = 0; j< colCountEven; evenIncrement(j))
       for (k = 0; k< pCountEven  ; evenIncrement(k))
          c[i + j] += a[i + k] * b[j + 2*k];
```

The index computations of $\overrightarrow{k}$ and $\overleftarrow{k}$ in the innermost loop above reduce to three RISC instructions, plus two to sum the matrix-element addresses. Although fast and constant time, this is still half-again what is available from column-major representation. With integer/floating processors tuned to column-major this difference may once have mattered but now it doesn't, with register operations so fast relative to memory access. It becomes important, though, for compilers to provide this arithmetic as loop control.

For three and higher dimensions, it seems best to implement arithmetic only for one type of dilated integer, and to translate other representations from it.

Dilated integers simplify input and output of Morton-ordered matrices in human-readable raster order [3]. Such convenience is not computationally significant because I/O delays dominate that indexing, but it may be politically important just to make this matrix representation accessible to the programmers who need to experiment with it.

3.2 SPACE AND BOUNDS

Theorem 5 tells how much address space an $m \times n$ matrix occupies, usually more than the mn positions containing data. As mentioned above and observed in experiments, if the difference, $\overleftarrow{m-1} + \overrightarrow{n-1} + 1 - mn$, is large then most of it will never move into faster memory. Moreover, the larger the difference, the more remote (and cheaper) will be the bulk of the addresses allocated. The size of that excess space for a rectangular matrix depends the number of elements and also on the aspect ratio between the census of rows and columns.

In block algorithms, cache hits make it better to iterate through Morton-order sequentially. For instance, if `b` is the initial index of a block from matrix A of size `n`=4^p that is to be zeroed, it is better to initialize it with a single, localized loop: `for (int i=0; i<n; i++) A[b+i]=0;` than to use column-major traversal.

Bounds checking on each row and column is elegant. First of all, Theorem 6 provides a fast check of either a Morton or an Ahnentafel index, `a`, against predilated row and column bounds:

```
if ( (a &oddBits)<rowCountOdd && (a &evenBits)<colCountEven ) ...
```

For Ahnentafel indices especially, the compiler can do even more. It will precompute two vectors of bounds on a row or column index at each level of the quadtree. The first, `rowBound[]`, is a bound on the perimeter of the matrix, to preclude access to southern and eastern padding. The second, `rowDense[]`, contains bounds on interior Ahnentafel indices that are north and west of this perimeter which, once passed, obviate the need for further bounds checking at any subtree/subblock. Allocate two vectors of size h to contain row bounds on the Ahnentafel indices at each level, where h is the height of the tree. For both, the hth entry is $\overleftarrow{r}$ where r is the number of rows. Thereafter,

```
rowBound[level-1] = rowBound[level]>>2;
rowDense[level-1] = (rowDense[level] ←1) >>2;
```

With c columns the column bounds are computed from $\overrightarrow{c}$ similarly, as even dilated integers; we usually test row and column limits simultaneously.

One might not be overly precise on Ahnentafel bounds at the perimeter. Because extra space is often allocated to the south and east anyway, it has been observed to be cheaper to round the actual bounds on the matrix up to, say, the next multiple of eight and then to treat the margins as "active" padding. For a recursive block algorithm, the alternative is to detect blocks of size four, two, and one where there will be too few operations; with the smallest block at order eight, operations on it can be dispatched as unconditional, straight-line, superscalar code. That is, we choose to treat padding in small marginal blocks as sentinels, initialized so each can participate in the gross algorithm without affecting its net result. Typically this is zero to the south or east, and the identity matrix to the southeast:

```
for (int jj=0; jj<n; jj= (jj-evenBits)&evenbits ) A[3*jj]=1;
```

The compiler also needs to use the algebra of dilated integers from Section 3.1 to unfold recursions and to unroll loops effectively; it becomes another kind of reduction in the strength of indexing.

Finally, symmetric matrices and matrix transpose are also easy with Morton or Ahnentafel indices. If `m` is either kind of index, then the index of its reflected element or (untransposed) block is quickly computed by exchanging its even and odd bits in a dosido: `( (m &evenBits) <<1) + ( (m &oddBits) >>1)`

4 CONCLUSION

We have already demonstrated vast improvements using both the cartesian and Ahentafel indexing schemes described here.

A good quadtree algorithm uses recursive descent on Ahnentafel indices, so there is no need to preselect a block size to fit one—or any—level of cache; the algorithm simply accelerates when a block fits [8]. None of the indices, themselves, need be stacked. Each can be shifted right to effect a stack pop or incremented to refer to a sibling. Still, innermost recursions need to be unfolded [2], just as the C compiler unrolls loops, in order to obtain straight-line code for superscalar processors. See [18] for more details; all the speed improvements are due to good locality [5].

The formalism presented here shows how to implement Morton-order matrices, with efficient algorithms for the group of dilated integers for the program that uses cartesian indexing, or with Ahnentafel indexing for the one that uses recursive descent on quadtrees. We have built a propotype compiler to translate C programs using row-major matrices and cartesian indices to Morton-order using dilated indices. We had already demonstrated the ease of tree-wise scheduling parallel processors in [7], and we continue to search for similar quadtree algorithms [17, 6].

It remains to close the gap between these efforts: on the one hand to improve compilers to optimize indexing on Morton-ordered matrices (*e.g.* unfolding Ahnentafel–controlled recursions and unrolling dilated-integer–controlled loops). And, on the other hand, to exercise Morton-order matrices under both styles, comparing performance of both families of algorithms by comingling them in a single environment as two libraries of interchangeable modules. Our underlying goal remains to uncover better algorithms that use more locality and balanced parallelism to solve matrix problems faster.

Acknowledgement: Thanks to Christian Weiss for helpful comments and special thanks to Hanan Samet who provided the early references.

References

1. J. Backus. The history of FORTRAN I, II, and III. In R. L. Wexelblat (ed.), *History of Program. Languages*, New York, Academic Press (1981), 25–45. Also preprinted in *SIGPLAN Not.* **13**, 8 (1978 August), 165–180.
2. R. M. Burstall & J. Darlington. A transformation system for developing recursive programs. *J.ACM* **24**, 1 (1997 January), 44–67.
3. F. W. Burton, V. J. Kollias, J. G. Kollias. Real-time raster-to-quadtree and quadtree-to-raster conversion algorithms with modest storage requirements. *Angew. Informatik* **4** (1986), 170–174.
4. H. G. Cragon A historical note on binary tree. *SIGARCH Comput. Archit. News* **18**, 4 (1990 December), 3.
5. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, & T. von Eicken. LogP: a practical model of parallel computation. *Commun. ACM* **39**, 11 (1996 November), 78–85.
6. J. Frens. *Matrix Factorization Using a Block-Recursive Structure and Block-Recursive Algorithms.* PhD dissertation, Indiana University, Bloomington (in progress).
7. J. Frens & D. S. Wise. Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code. *Proc. 1997 ACM Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7 (1997 July), 206–216.
8. M. Frigo, C. E. Leiserson, H. Prokop, & S. Ramachandran. Cache-oblivious algorithms, Extended abstract. Lab for Computer Science, M.I.T., Cambridge, MA (1999 May). `http://supertech.lcs.mit.edu/cilk/papers/abstracts/FrigoLePr99.html`
9. Y. C. Hu, S. L. Johnsson & S.H. Teng. High performance Fortran for highly irregular problems. *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not.* **32**, 7 (1977 July), 13–24.
10. D. E. Knuth. *The Art of Computer Programming* **I,** *Fundamental Algorithms* (3rd ed.), Reading, MA: Addison-Wesley, (1997).
11. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Ottawa, Ontario: IBM Ltd. (1966 March 1).
12. J. A. Orenstein & T. H. Merrett. A class of data structures for associative searching. *Proc. 3rd ACM SIGACT–SIGMOD Symp. on Princ. of Database Systems* (1984), 181–190.
13. H. Samet. *The Design and Analysis of Spatial Data Structures* Reading, MA: Addison-Wesley, (1990), §2.7.
14. G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.* **55**, 3 (1992 May), 221-230.
15. K. D. Tocher. The application of automatic computers to sampling experiments. *J. Roy. Statist. Soc. Ser. B* **16**, 1 (1954), 39–61.
16. M. S. Warren. & J. K. Salmon. A parallel hashed oct-tree N-body problem. *Proc. Supercomputing '93.* Los Alamitos, CA: IEEE Computer Society Press (1993), 12–21.
17. D. S. Wise. Undulant block elimination and integer-preserving matrix inversion. *Sci. Comput. Program.* **33**, 1 (1999 January), 29–85.
`http://www.cs.indiana.edu/ftp/techreports/TR418.html`
18. D. S. Wise & J. Frens. Morton-order matrices deserve compilers' support. Technical Report 533, Computer Science Dept, Indiana University (1999 November).
`http://www.cs.indiana.edu/ftp/techreports/TR533.html`

An Efficient Parallel Linear Solver with a Cascadic Conjugate Gradient Method: Experience with Reality

Peter Gottschling[1]* and Wolfgang E. Nagel[2]

[1] GMD FIRST, Berlin, Germany,
pg@first.gmd.de
[2] Center for High Performance Computing (ZHR),
Dresden University of Technology, Germany,
nagel@zhr.tu-dresden.de

Abstract. To solve large systems of linear equations with sparse matrices in parallel, there are three factors that contribute to the computing time: the numerical efficiency, the floating point performance, and the scalability. In this paper, we mainly consider the floating point performance. For large linear systems, multi-level techniques, like the cascadic conjugate gradient method (CCG), require significantly less operations than single-level methods. On the other hand, they are considered less efficient with regard to performance and limited in parallelization. Therefore, to achieve an efficient, massively parallel multi-level solver, we used the fastest available communication and revised the whole computation. The performance improvements led to a parallel solver which is able to solve a linear system with more than 16 million unknowns in 0.77 seconds on 256 PEs of Cray T3E. This corresponds to an overall performance of 10.34 GFLOPS.

Keywords. Floating Point Performance, RISC Processors, Matrix Sparsity Pattern, Cascadic Conjugate Gradient Method

1 Introduction

The solution of large systems of linear equations with sparse matrices plays an important role in many simulation applications and in some of the so-called 'grand challenge' problems. Three components can be identified which determine the necessary time to solve a linear system in parallel: the numerical efficiency, the floating point performance, and the scalability.

First of all, one should examine whether it is possible to apply a multi-level solver – like multigrid or cascadic conjugate gradient method – to the investigated problem. Single-level solvers – like Gauss-Seidel – most often converge poorly for large linear systems. Though the floating point performance and the

* Part of the work was done while the author was a research associate at ZHR in 1999.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 784–794, 2000.
© Springer-Verlag Berlin Heidelberg 2000

parallel efficiency is sometimes better, this cannot compensate for the numerical deficiencies. The usually better parallelism is explained by the larger ratio between the number of operations and the size of transferred data between the processors because multi-level methods work on several linear systems with different dimensions.

Multi-level techniques use single-level methods on the individual levels. Therefore, the floating point performance on several levels is similar to that of the single-level techniques used. On the smaller linear systems, the performance is sometimes even higher because of better cache reuse. On the other hand, the additional operations on multi-level techniques – the transfer operations between the grids – perform poorly due to indirect and irregular memory accesses. Nevertheless, the computing time of the transfer operations is usually quite short. For that reason, a multi-level technique can sometimes perform similarly as the containing single-level method.

The systems of linear equations, which we considered in our investigations, originate from the discretization of the ground-water flow equation. The strong variation of the parameters of this partial differential equation causes strongly varying coefficients in the matrix of the linear system. Despite the variation of the coefficients and the largeness of the linear system, the cascadic conjugate gradient method with an algebraically generated hierarchy of linear systems enables good convergence [5].

The paper is organized as follows. In section 2, we consider different types of sparse matrices. For these matrix types, the counter-movement of the applicability of discretization schemes and the possibilities of performance tuning is shown. The communication expense is covered in the subsequent section. Section 4 presents the optimization targets for the arithmetic part used in the parallel solver. Different implementations of the matrix vector multiplication are compared in section 5. The last section describes the optimization of the conjugate gradient method.

2 Sparsity Patterns of Matrices

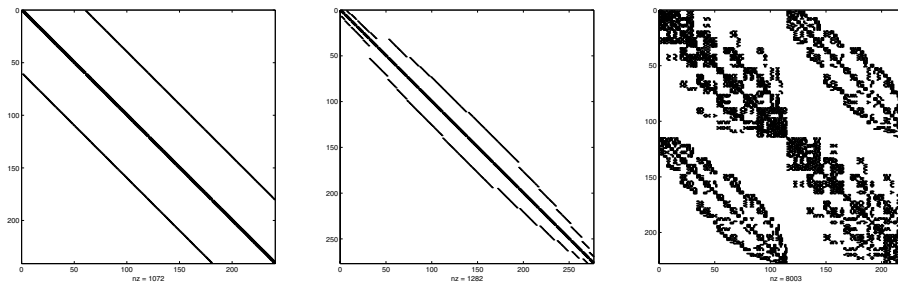
Often, systems of linear equations with sparse matrices originate from discretized partial differential equations. The type of discretization determines the sparsity pattern of the matrix. In this paper, we distinguish three types of matrices.

Structured Matrices: These matrices (fig. 1a) are characterized by a set of constants $C = \{c_1, c_2, \ldots c_m\}$ so that $j - i \notin C \rightarrow a_{ij} = 0$. This means that only matrix elements with a certain distance from the diagonal can be non-zero. Matrices of this form arise in equidistant discretizations of rectangular or cuboid domains. Matrix vector products with this type can be programmed with simple loops using constant offsets. Therefore, different optimizations like loop unrolling and blocking (cf. [4]), are applicable.

Locally Structured Matrices: For the second type of sparsity, the expression 'locally structured' matrices (fig. 1b) is introduced. Here, several sets

of differences $C_1, C_2, \ldots$ can be defined where each set is valid for a certain interval of lines. The sparsity pattern can be expressed in an implementation by structural specifications that correspond with these intervals of matrix lines. This representation at least allows floating point optimizations within the intervals. Locally structured matrices originate, for instance, from the equidistant discretization of domains with irregular borders.

Unstructured Matrices: The most general kind of sparse matrices are unstructured matrices (fig. 1c, from [6]). No assumptions about the sparsity pattern are made here. Thus, arbitrary discretizations are permitted. On the other hand, each matrix element must be treated separately in a matrix vector multiplication and the performance is lower for that reason. Nevertheless, discretizations that are adapted to the problem are often necessary and the lower speed is justified by a significant reduction of the equation size.



(a) structured (b) locally struct. (c) unstructured

Fig. 1. Types of sparsity patterns

In our work, we consider locally structured matrices because of their importance in ground-water flow simulations. The two other matrix types are used in numerous other projects (structured matrices e.g. in [1] and unstructured ones e.g. in [10]).

3 Communication Expense

The computation of one step of the conjugate gradient method involves one exchange of the inner borders in the matrix vector multiplication and two reductions in dot products. Further exchanges of the inner borders are necessary in some preconditionings (e.g. incomplete Cholesky factorization). Moreover, the implementation of the termination criterion requires an additional reduction unless the values of the conjugate gradient method (CG) are used (cf. [2]).

The data dependencies in the conjugate gradient method allow the simultaneous reduction of the termination criterion and of the first dot product in the next iteration step of the CG method. Since the communication latency is

rather large compared to the bandwidth on every parallel computer and computer network the global reduction of two values takes roughly the same time as the reduction of one value.

To minimize the expense of the partitioning, the domain (on the fine grid) was decomposed by coordinate section in $P_x \times P_y$ subdomains on $P = P_x \cdot P_y$ processors. The decomposition was passed to the coarser grid where the boundaries were slightly adapted to conserve the load balance (cf. [5]).

A significant decrease of the communication time can sometimes be established by replacing portable communication procedures with proprietary ones. Figure 2 shows the time line, visualized by VAMPIR [7], of two iterations of the CG-method on a linear system with about 16,000 unknowns on Cray T3E. In this implementation based on the MPI library, the interprocessor communication, represented by lines, is the dominant part. Implementing the communication with equivalent shmem functions (shmem_double_get and shmem_double_sum_to_all) clearly shortens the communication time so that the execution time of one iteration is reduced from 1.437 ms to 0.632 ms (figure 3).

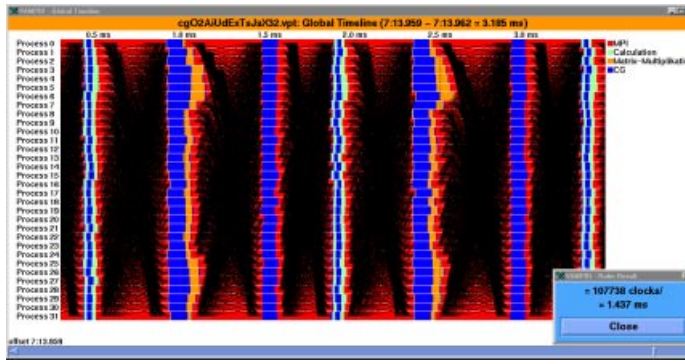


Fig. 2. Two iteration steps with MPI communication

Although solving a linear system with 16,000 unknowns is not very interesting to parallel computing, there are two reasons for accelerating the communication. Firstly, the communication becomes more important for increasing numbers of processors, even for large problems, and secondly, multilevel solvers spend most of their time on communication (delay mainly due to latency issues) while solving the coarse grid equations. Since we are interested in multilevel solvers on many processors, an optimized communication is very important.

4 Optimization Targets to Improve the Floating Point Performance on RISC Processors

To increase the floating point performance of a RISC processor for our application, we followed four goals. We focused on the DEC Alpha 21164 of Cray T3E,

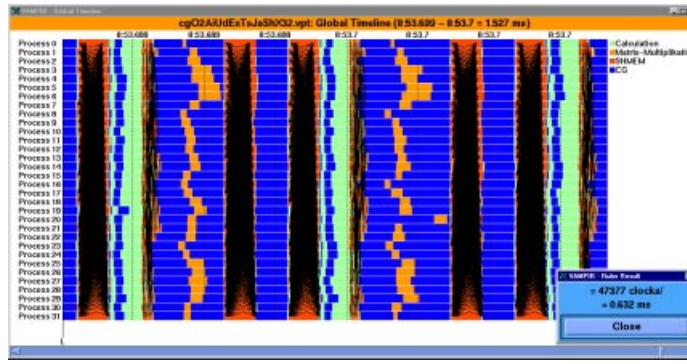


Fig. 3. Two corresponding iterations with shmem communication

nevertheless, the optimization steps should be helpful on other superscalar RISC processors, too.

Decrease of the Memory Accesses by Cache Reuse: The main bottleneck of fast RISC processors is the rather slow main memory access. On Cray T3E, for instance, processors with 300-600 MHz and 600-1200 MFLOPS face main memories with 75 MHz. To yield floating point performance near peak performance, it is necessary (but not sufficient) to reuse data in registers or in the primary cache as often as possible so that memory accesses are not relevant to the execution time. Seidl [9] has shown by examples that the acceleration of algorithms can be predetermined from the reduced probability of memory accesses.

Consecutive Memory Accesses in Increasing Order: Loading a data item into the cache is realized by loading a certain amount of memory, called cache line, which is typically larger than the data item itself. If referred subsequently, the next data items are already in the cache with some probability. On Cray T3E, additional benefit can result from the stream buffers. This is a mechanism that looks for increasing memory accesses and, if recognized, loads successive cache lines into a special register where they can be rapidly loaded into the second level cache (cf. [8]). Operations on large vectors are, from the authors' experience, typically computed twice as fast with stream buffers.

Independent Operations: Implementations with many independent operations allow the efficient use of multiple pipelines. The number of independent operations is very often increased by loop unrolling.

Reduction of Divide Operations: The divisions of floating point numbers are not as fast as additions and multiplications on most processors. In addition, they cause pipeline blocking on the DEC Alpha 21164.

5 Matrix Vector Multiplication

The importance of the matrix vector multiplication is based on the fact that in our solver half of the floating point operations are performed in this section.

In the original implementation, the multiplication was calculated in two steps. At first, the result vector was initialized with the product of the input vector and the diagonal of the matrix. Then, the components of the result vector were incremented by non-zero matrix entries outside the diagonal multiplied with the elements of the input vector. In this way, sub-vectors of maximal length were used and the loop overhead was minimized.

As an example, we considered a multiplication with a vector containing about 126,000 elements and an appropriate matrix, so that about 1,134,000 floating point operations had to be executed. The original implementation required 34.9 ms to compute the multiplication on the 600 MHz processor applying stream buffers. This corresponds to 32.5 MFLOPS.

While in the second section approximately 8 floating operations per unknown were computed in relation to 4 to 6 memory accesses (depending on the extension and form of the domain and the size of the second level cache), there are three memory accesses per unknown with only one operation performed in the first section. This unfavorable ratio between operations and memory accesses resulted in a very slow computation. Once more looking at Cray T3E, on the 600 MHz processor the initialization step performed with 8.4 MFLOPS without stream buffers where the performance was only augmented to 18.3 MFLOPS by using the stream buffers. To emphasize the importance of the memory bandwidth, the same operation was performed with 7.5 MFLOPS and 15 MFLOPS respectively on a 300 MHz processor.

To avoid this slow computation, the initialization of the output vector was included in the incrementing step. The product of the diagonal and the input vector was then only computed for those elements where the output vector was incremented in the next moment. This modified implementation saved 2 memory accesses per unknown for loading the vectors (unless the sub-vectors involved in the incrementing section were too large for the cache) but additional overhead was produced to control which components of the result vector must be initialized and due to dividing the initialization into several sections. On the example problem, this implementation took 26.9 ms for the multiplication (42.2 MFLOPS).

Experiments with structured matrices have shown that matrix vector multiplications were significantly faster if the elements of the result vector were computed explicitly (`v[i]= a[i][i]*q[i] + a[i][j]*q[j] +` $\cdots$) instead of by incrementing as described above. For locally structured matrices, it is more complicated to apply the explicit calculation. In comparison with the former implementation, the loops are shorter and they need more preparations. Altogether, the loop overhead is more than doubled, although the computing time is decreased by 59.5 per cent to 14.1 ms on the example problem (80.2 MFLOPS).

Another performance optimization is applicable if the multiplication is part of the conjugate gradient method. There, the dot product of the input and the output vector is used. Since two memory accesses are necessary per floating point

operation this calculation is slow (there is a bottleneck on the scalar value, too). For the considered vector size, the dot product requires 5.1 ms.

Including this dot product in the matrix vector multiplication, as proposed in [3], saves the memory accesses. Consequently, most of the computing time for the dot product is saved (the scalar value is less critical here because several floating point operations are performed between two increments).

In a parallel implementation, attention has to be paid to the inner boundaries. To enable a multiplication with a symmetric matrix, vector components `q[j]` that are assigned to other processors are considered in an extra computation (`v[i]+= a[i][j]*q[j]`). In this section, the dot product is calculated as `dot+= a[i][j]*q[j]*q[i]` while the computation with the symmetric part of the matrix is `dot+= v[i]*q[i]`. Altogether, the execution time of the combined computation was 14.7 ms which corresponds to 93.9 MFLOPS.

6 Iteration Steps of the Conjugate Gradient Method

The conjugate gradient method itself only consists of vector operations. Since vector operations are characterized by a poor ratio between the number of floating point operations and the number of memory accesses, the performance is rather low. To improve this ratio, the vector operations have to be combined to compute as many floating point operations as possible on a vector component while it resides in the cache. Whereas calculations depending on a vector can start as soon as a part of the vector is available, calculations depending on a scalar value resulting from a vector reduction must wait until the reduction is finished.

Although the conjugate gradient method is well known, we present the iteration step as a C++ program in table 1 for a better illustration. The startup phase is omitted because it does not provide further optimization opportunities. The extension of the CG method to the cascadic conjugate gradient method is quite simple. Starting on the coarsest grid, the CG method is computed on every grid. When the termination criterion is fulfilled on a certain grid the vector x is interpolated to the next finest grid and is used as an initial guess for the CG method on that grid.

The program can be implemented in this form by using templates in C++. The use of templates is critical because each operator is computed separately. For this reason, temporary vectors are required, which produce overhead for additional memory allocations and memory accesses, unless advanced numerical template libraries like Blitz++ [11] are used.

In the calculation of dot products, the scalar value represents a bottleneck for superscalar processors. Since the addition is associative (in exact arithmetic), the products of the vector components can be summed into several values within the loop and added at the end. For the vector size considered in the previous section, the computing time was reduced from 5.1 ms to 3.1 ms, which corresponds to an increase of the performance from 49.2 MFLOPS to 77.8 MFLOPS. In our

```
vector<double>      x, v, q, r, w;
double              alpha, delta, gamma, gamma_old;

int cg_iteration ()
{ double     delta_local, gamma_local;

  q= (gamma/gamma_old) * q + w;
  exchange_inner_borders (q);          // requires communication
  v= a_mult (q);
  delta_local= dot (v, q);
  delta= reduce (delta_local);         // requires communication
  alpha= gamma / delta;
  x+= alpha * q;
  r-= alpha * v;
  w= thepc→f (r);                      // chosen preconditioning (possibly requires comm.)
  gamma_local= dot (w, r);
  gamma= reduce (gamma_local);         // requires communication
  return thepc→f ();                   // chosen termination criterion (usually requires comm.)
}
```

Table 1. Iteration of the conjugate gradient method

investigations, changing the order of the summation did not noticeably influence the exactness of the floating point arithmetic.

Inlining a particular preconditioning and a particular termination criterion saves the function calls and allows further optimization. For the investigated systems of linear equations, the diagonal preconditioning $w = D^{-1}r$ represented the best compromise between the numerical properties and the computational and communicational expense. Among different termination criteria, it has been shown that for equations with strongly varying coefficients, the diagonally preconditioned residual $D^{-1}r$ enables the best error estimation. So, the commitment to this combination permits the elimination of redundant calculations.

Since the element-wise division is rather slow and the vector components are divided by constant values, it is worthwhile to store the inverse of the diagonal matrix. An element-wise multiplication then replaces the element-wise division at the price of an extra vector and some additional computation before starting the linear solver.

To reduce the number of memory accesses, the calculations of `r-= alpha * v`, `w= adinv * r`, `dot (w, r)` and `dot (w, w)` can be combined in one loop, where `adinv` is a vector with `adinv`$[i] = 1/a_{ii}$. Of course, this loop can be unrolled, too. With the simultaneous reduction of the local computations of `dot (w, r)` and `dot (w, w)` and the modifications described above, the iteration of the conjugate gradient method looks as shown in table 2.

As an example, a linear system with more than 16 million unknowns was solved on 32 processors. Here, the execution time of a single iteration step on the finest level was decreased from 432 ms in the original version to 206 ms in the accelerated one. Furthermore, it has been shown that the variations of the computing time between the different processors gain in significance due to the performance tuning. Although the number of operations are equally distributed among the processors, the computing time varies noticeably. As a consequence,

```
int     nupo;     // number of points corresponds to vector size

int cg_iteration_jacobi ()
{ double     delta_s, delta_a, delta_local, stop_gamma_local [2], stop_gamma [2],
             s0, s1, s2, s3, g0, g1, g2, g3, tmp0, tmp1, tmp2, tmp3;
  int        i, nupo4= nupo ≫ 2 ≪ 2;

  scadd (gamma/gamma_old, q, w);
  exchange_inner_borders (q);                                  // requires communication
  delta_local= a_mult (v, q);
  delta= reduce (delta_local);                                 // requires communication
  alpha= gamma / delta; scadd (x, alpha, q);
  s0= s1= s2= s3= g0= g1= g2= g3= 0.0;
  for (i= 0; i < nupo4; i+= 4)
    {tmp0= (r [i]-= alpha ∗ v [i]) ∗ adinv [i]; s0+= tmp0 ∗ tmp0; g0+= r [i] ∗ tmp0;
    tmp1= (r [i+1]-= alpha ∗ v [i+1]) ∗ adinv [i+1]; s1+= tmp1 ∗ tmp1; g1+= r [i+1] ∗ tmp1;
    tmp2= (r [i+2]-= alpha ∗ v [i+2]) ∗ adinv [i+2]; s2+= tmp2 ∗ tmp2; g2+= r [i+2] ∗ tmp2;
    tmp3= (r [i+3]-= alpha ∗ v [i+3]) ∗ adinv [i+3]; s3+= tmp3 ∗ tmp3; g3+= r [i+3] ∗ tmp3;
    w [i]= tmp0; w [i+1]= tmp1; w [i+2]= tmp2; w [i+3]= tmp3;}
  for (i= nupo4; i < nupo; i++)
    {w [i]= tmp0= (r [i]-= alpha ∗ v [i]) ∗ adinv [i]; s0+= tmp0 ∗ tmp0; g0+= r [i] ∗ tmp0;}
  stop_gamma_local [0]= s0 + s1 + s2 + s3; stop_gamma_local [1]= g0 + g1 + g2 + g3;
  reduce2 (stop_gamma_local, stop_gamma);                      // requires communication
  gamma_old= gamma; gamma= stop_gamma [1];
  return sqrt (stop_gamma [0]) < thetc→epsilon;
}
```

Table 2. Iteration of the specialized and accelerated version

the parallel execution time was affected more by the loss of synchronism than by the communication expense on Cray T3E.

On two processors, where the balance of the computing time and the communication are less significant, the accelerated implementation achieved a performance of 66.2 MFLOPS per processor. On 256 processors,[1] the performance per processor was more than 40 MFLOPS, leading to an overall performance of 10.34 GFLOPS. To solve the linear system with 16 million unknowns, the solver based on the cascadic conjugate gradient method and on algebraically generated coarse grid equations required 9 iterations of the CG method on the finest grid and 15 iterations on the other grids [5]. So, the linear system was solved in 0.77 seconds. Since many simulations of physical processes – like ground-water flow – are based on the solution of many large linear systems, a fast parallel multi-level solver can save a lot of computing time. On the other hand, the acceleration of the solver may also be used to increase the simulation complexity in order to improve the simulation results.

7 Conclusion

Although on modern computer architectures the memory bandwidth is still by far too low with regard to the processor speed, this relation is expected to get worse in the near future (cf. [4, p. 34]). For this reason, the primary performance optimization target is reducing the number of main memory accesses. Therefore,

[1] The authors would like to thank Prof. Hoßfeld from John von Neumann-Institute for Computing (NIC-ZAM) for providing capacity on Cray T3E.

the computations have to be reordered so that as many operations as possible are performed on a data item while it resides in cache.

Fortunately, there is a relatively small kernel in many numerical applications where most of the computing time is spent. In this case, the performance tuning efforts can be restricted to this kernel so that the expense to improve the performance is usually low compared with the expense for the program development.

In scientific applications described by partial differential equations most of the execution time is usually spent on the solution of linear systems. Therefore, the acceleration of the linear solver can yield a great profit. First of all, it should be examined whether a multi-level method can be applied to the respective type of the linear system. In this case, the number of operations can often be decreased by several orders of magnitude by using a different algorithm. For the considered linear systems, which originate from the ground-water flow equation, the difficulty lies in the strong variation of the coefficients (up to 10^8). Nevertheless, the cascadic conjugate gradient method with algebraically generated coarse grid equations converged well for the examined linear systems.

To implement the parallel CCG solver efficiently, the whole iteration step of the CG method was optimized with regard to the performance, including the preconditioning and the termination criterion. A special matrix type is introduced which allows the demanded applicability of discretization schemes and provides more performance optimization possibilities than unstructured matrices. In addition, the communication time was significantly shortened by changing from the portable MPI library to the proprietary shmem library. So, even multilevel solvers can work efficiently on up to several hundred PEs on Cray T3E. Altogether, the fast communication, the high floating point performance and the good convergence enabled the solution of a linear system with over 16 million unknowns in less than a second.

References

[1] Manfred Alef. Implementation of a multigrid algorithm on suprenum and other systems. *Parallel Computing*, 20:1547–1557, 1994.

[2] Peter Deuflhard. Cascadic conjugate gradient methods for elliptic partial differential equations. algorithm and numerical results. In D. Keyes and J. Xu, editors, *Proc. of the 7th Int. Conf. on Domain Decomp. Methods 1993*, pages 29–42, 1994.

[3] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998.

[4] Kevin Dowd and Charles R. Severance. *High Performance Computing*. O'Reilly, Sebastopol, second edition, 1998.

[5] Peter Gottschling. Efficient parallel computation of the discretized ground-water flow equation with algebraic multigrid methods (in german). Phd thesis, TU Dresden, in preparation.

[6] Uwe Lehmann. Festigkeitsuntersuchung von Eisenbahnfahrwegen auf Parallelrechnern. Diplom, TU Dresden, 1999.

[7] W.E. Nagel and A. Arnold. Performance visualization of environment. Technical report, Forschungszentrum Jülich, 1995. http://www.fz-juelich.de/zam/PT/ReDec/SoftTools/PARtools/PARvis.html.

[8] Wolfgang E. Nagel. The new massively parallel computer cray t3e in spring 1996: Experience with virginity (in german). In Hans-Werner Meuer, editor, *Supercomputer 1996*, pages 92–107. K. G. Saur, 1996.
[9] Stephan Seidl. Code crumpling: A straight technique to improve loop performance on cache based systems. In *Proc. 5th Eur. SGI/Cray MPP Workshop*, 1999.
[10] J. Stiller, K. Boryczko, and W.E. Nagel. A new approach for parallel multigrid adaption. In *Proc. 9th SIAM Conf. on Par. Proc. for Sci. Comp.*, 1999.
[11] Todd Veldhuizen et al. *Blitz++*. www://oonumerics.org/blitz/.

A Fast Solver for Convection Diffusion Equations Based on Nested Dissection with Incomplete Elimination

Michael Bader and Christoph Zenger

TU München, Lehrstuhl für Informatik V,
D-80290 München, Germany
{bader,zenger}@in.tum.de
http://www5.in.tum.de/

Abstract. We present an approach for the efficient parallel solution of convection diffusion equations. Based on iterative nested dissection techniques [1] we extended these existing iterative algorithms to a solver based on nested dissection with incomplete elimination of the unknowns. Our elimination strategy is derived from physical properties of the convection diffusion equation, but is independent of the actual discretized operator. The resulting algorithm has a memory requirement that grows linearly with the number of unknowns. This also holds for the computational cost of the setup of the nested dissection structure and the individual relaxation cycles. We present numerical examples that indicate that the number of iterations needed to solve a convection diffusion equation grows only slightly with the number of unknowns, but is widely independent of the type and strength of the convection field.

1 Introduction

We are searching for an efficient parallel solver for the linear systems of equations arising from the discretization of the convection diffusion equation

$$-\triangle u + a(x,y) \cdot \nabla u = f. \tag{1}$$

In this paper, we focus on standard finite difference or finite element discretizations on rectangular Cartesian grids resulting in the standard five or nine point discretization stencils.

To obtain a truly efficient solver for convection diffusion equations one has to overcome three main problems. The performance of the solver should be independent of the convection field $a(x,y)$. The solver should be able to treat arbitrary geometries of the computational domain with equal efficiency. Finally, it should be possible to produce efficient parallel implementations of the solver to take optimal advantage of high performance computers.

Solvers based on geometric multigrid methods are usually quite easy to parallelize due to their structured coarse grids. On the negative side they often have difficulties in treating complicated geometries. Moreover, they often require a

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 795–805, 2000.
© Springer-Verlag Berlin Heidelberg 2000

special treatment of curved or circular flow fields, which sometimes seems to impair the parallelization properties.
Algebraic multigrid (AMG) methods are usually very robust with respect to the strength or type of the convection or even the geometry. As AMG chooses its coarse grids according to the operator and the geometry, it often produces excellent convergence results. On the other hand, the automatic grid generation often makes it difficult to produce efficient parallel implementations of the solvers. This is especially true for the construction of the different coarse grids itself, because the most commonly used strategy [5] for the coarse grid selection is an inherently sequential process.

Our goal is therefore to find a fast (i.e. with multigrid or multigrid-like performance) method that is easy to parallelize and allows the treatment of an arbitrary geometry of the computational domain. We base our approach on previous work by Hüttl [1] and Ebner [2], whose algorithms combine ideas from domain decomposition and nested dissection techniques to iterative methods based on recursive substructuring of the computational domain.

After giving a short comprehension of these techniques in section 2, we will, in section 3, present our extension of these approaches by an incomplete elimination of the most significant couplings in the system matrices. Finally, in section 4 we will show some numerical examples that indicate the promising potential of our approach.

2 The Nested Dissection Approach

2.1 Nested Dissection as a Direct Solver

The nested dissection method was introduced by George [4] as a direct solver for the sparse linear systems of equations resulting from the discretization of PDEs with finite elements. It it based on the recursive substructuring of the computational domain by which it minimizes the fill-in that usually occurs during the solution process. Throughout this paper we will divide the nested dissection algorithm into three passes: the recursive substructuring, the static condensation, and the solution.

Pass 1: Recursive Substructuring

In the first (top-down) pass the computational domain is recursively divided into two or more subdomains that are connected via a *separator*. As shown in figure 1, we will use a separation by alternate bisection throughout this paper. Compared to substructuring in four (or even more) subdomains the alternate bisection produces linear systems of equations with a slightly smaller number of unknowns, which gives advantages in parallelization.

On each subdomain, we classify the unknowns into the set $\mathcal{I}$ of the inner unknowns and the set $\mathcal{E}$ of the outer unknowns. The inner unknowns are those unknowns on the separator that do not belong to the separator of a parent

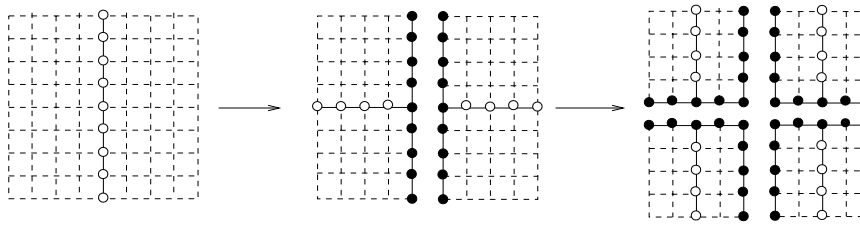


Fig. 1. Recursive substructuring by alternate bisection

domain. The outer unknowns lie on the border of the subdomain and are exactly those unknowns that will become separator unknowns on a parent domain. The other unknowns inside the subdomain can be ignored, as their couplings with inner and outer unknowns are eliminated by the static condensation (see pass 2) on the lower levels. Figure 1 illustrates this classification by painting the inner unknowns as white circles and the outer unknowns as black circles.

Pass 2: Static Condensation

The static condensation pass is a bottom-up process and computes the local systems of equations for each subdomain. On the finest level — i.e. when the subdomain has only 2×2 or 3×3 unknowns and is not further divided — the system of equations can be taken directly from the discretization. On the higher levels the system is computed from the local systems of the two subdomains. The first step is to assemble and renumber the system matrix:

$$V^T \begin{pmatrix} K_1 & 0 \\ 0 & K_2 \end{pmatrix} V =: \begin{pmatrix} K_{\mathcal{E}\mathcal{E}} & K_{\mathcal{E}\mathcal{I}} \\ K_{\mathcal{I}\mathcal{E}} & K_{\mathcal{I}\mathcal{I}} \end{pmatrix}. \tag{2}$$

The operator V combines the system matrices K_1 and K_2 that are taken from the two subdomains. The unknowns on the separator belong to both subdomains, thus the corresponding matrix lines are simply added up. The renumbering is such that the outer and inner unknowns form separate matrix blocks $K_{\mathcal{E}\mathcal{E}}$ and $K_{\mathcal{I}\mathcal{I}}$ to enable the following block elimination.

In this second step the couplings between the inner and the outer unknowns are eliminated by computing the Schur complement

$$\begin{pmatrix} \mathrm{Id} & -K_{\mathcal{E}\mathcal{I}}K_{\mathcal{I}\mathcal{I}}^{-1} \\ 0 & \mathrm{Id} \end{pmatrix} \begin{pmatrix} K_{\mathcal{E}\mathcal{E}} & K_{\mathcal{E}\mathcal{I}} \\ K_{\mathcal{I}\mathcal{E}} & K_{\mathcal{I}\mathcal{I}} \end{pmatrix} \begin{pmatrix} \mathrm{Id} & 0 \\ -K_{\mathcal{I}\mathcal{I}}^{-1}K_{\mathcal{I}\mathcal{E}} & \mathrm{Id} \end{pmatrix} = \begin{pmatrix} \widetilde{K}_{\mathcal{E}\mathcal{E}} & 0 \\ 0 & K_{\mathcal{I}\mathcal{I}} \end{pmatrix} \tag{3}$$

$$\text{where} \quad \widetilde{K}_{\mathcal{E}\mathcal{E}} = K_{\mathcal{E}\mathcal{E}} - K_{\mathcal{E}\mathcal{I}} \cdot K_{\mathcal{I}\mathcal{I}}^{-1} \cdot K_{\mathcal{I}\mathcal{E}}.$$

Of course, the right-hand sides have to be treated accordingly.

Pass 3: Solution

The solution itself is again a top-down process. Starting with the separator of the whole computational domain, the values of the separator unknowns are computed recursively from the local system of equations on each subdomain.

2.2 Iterative Versions of the Nested Dissection Method

In the case of a two-dimensional PDE using standard discretization with a five- or nine-point stencil, computing the direct solution of the resulting linear system of equations (with $N = n \times n$ unknowns) needs $O(N^{\frac{3}{2}})$ operations and requires $O(N \log N)$ memory [4]. While one can sometimes put up with the $O(N^{\frac{3}{2}})$ operation count and the resulting increased computing time, shortness of memory often inhibits simulations altogether as the required resolution exceeds the memory capacity. However, the nested dissection method is often used in industrial codes for reason of reliability and robustness. For Poisson type equations, Hüttl [1] and Ebner [2] introduced iterative versions of the nested dissection method that have a memory requirement that grows only linearly with the number of unknowns.

The main differences between an iterative and a direct solver occur in the condensation and the solution pass. While the assembly part of the condensation pass mainly stays the same, the elimination of the separator couplings is left out entirely or at least greatly reduced in an iterative solver. The single top-down solution pass is replaced by a series of iteration cycles. Each of those cycles consists of a bottom-up pass that transports the current residual, like the right-hand side in the condensation pass, from the finest level subdomains to the higher levels. The second part of the iteration cycle is the actual solution, where Gauss-Seidel- or Jacobi-relaxation is used for the computation of the separator unknowns. Figure 2 illustrates the complete sequence of the different passes during the iterative nested dissection.

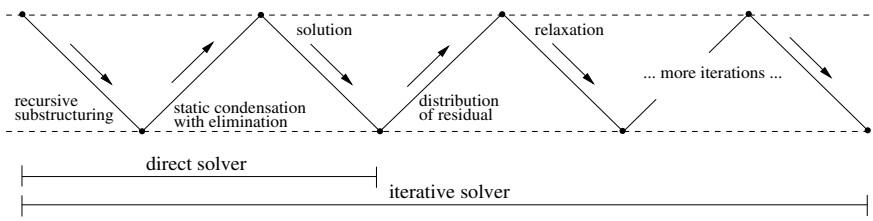


Fig. 2. The different passes of the iterative nested dissection algorithm

To get an acceptable speed of convergence, the systems of equations are preconditioned, for example by means of hierarchical bases [6]. The resulting algorithms reduce the number of operations to $O(N \log N)$ and the memory requirements back to $O(N)$. The convergence factor of such an iterative nested dissection method can be further improved by eliminating at least some of the couplings between the hierarchically highest unknowns in the local systems of equations [2]. Figure 3 shows the general structure of the resulting algorithm, which will be the basis for the algorithm introduced in section 3.

Pass 2: static condensation (set-up phase)

(S1) $K' = V^T \begin{pmatrix} K_1 & 0 \\ 0 & K_2 \end{pmatrix} V$	assemble system matrix from subdomains
(S2) $\bar{K} = H^T K' H$	hierarchical transformation
(S3) $K = L^{-1} \bar{K} R^{-1}$	partial elimination of the separator couplings, the elimination matrices L and R have to be stored

Pass 3: iteration cycle (bottom-up part)

(U1) $r' = V^T \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}$	assemble local residual from subdomains
(U2) $\bar{r} = H^T r'$	hierarchical transformation
(U3) $r = L^{-1} \bar{r}$	right-hand side of elimination

Pass 3: iteration cycle (top-down part)

(D1) $\hat{u}_S^{[it]} = \omega \mathrm{diag}(K)^{-1} r$	relaxation step (here: weighted Jacobi)
(D2) $\bar{u} = R^{-1} \hat{u}^{[it]}$	revert elimination
(D3) $u = H \bar{u}$	hierarchical transformation
(D4) $\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = V^T u$	distribute solution to subdomains

Fig. 3. Iterative Nested Dissection Algorithm: steps (S3), (U3) and (D2) are only needed if partial elimination is used

2.3 Parallel, Iterative Nested Dissection

The parallelization of iterative nested dissection algorithms like the one described in figure 3 was analysed in depth by Hüttl [1] and Ebner [3]. Figure 4 shows a typical distribution of the subdomains to different processors. Obviously, the tree structure of the subdomains and the simple bottom-up/top-down structure of the several computational cycles enable the parallel processing of all subdomains that are on the same level. However, the different levels have to be executed sequentially.

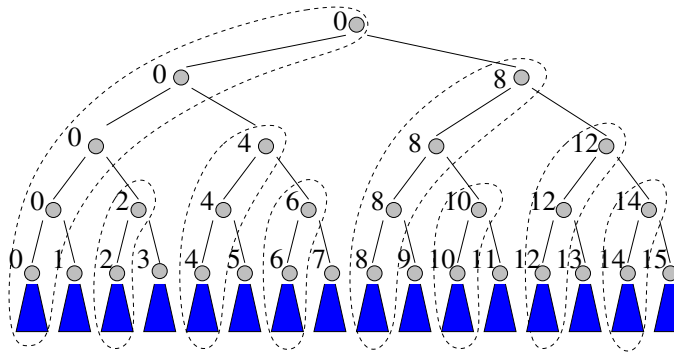


Fig. 4. Distribution of subdomains to 16 processors

The computations on each subdomain are usually not parallelized, so the size of these sequential subproblems, which is mainly determined by the size of the separator, should be kept small. We already mentioned that, in this context, the alternate bisection used in the recursive substructuring pass gives some advantage over other substructuring approaches. When using Jacobi relaxation for solution, the memory requirement on each subdomain is only linearly dependent on the number of unknowns on the separator. Therefore, these methods are particularly well suited for architectures with distributed memory.

Compared to other types of solvers (geometric multigrid, AMG, etc.), one should also emphasize that all three passes of the algorithm have equally good parallelization properties.

3 Nested Dissection with Incomplete Elimination

While the algorithms discussed in the previous section give good results for the solution of diffusion type equations, this no longer holds when they are applied to convection diffusion equations. In this section, we want to propose an approach which extends the existing iterative algorithms, such that convection diffusion equations can be treated efficiently as well.

When solving simple diffusion type equations, the elimination of the couplings between inner and outer unknowns can well be left out entirely, yet the convergence rates can be improved by eliminating just a small, fixed number of only the strongest couplings. For example, it is sufficient to eliminate the couplings between the unknown in the middle of the separator and those four unknowns that lie on the corners of the subdomain. However, for the convection diffusion case, eliminating a fixed number of couplings on each subdomain is no longer sufficient to achieve fast convergence.

In general, we can expect that the more couplings we eliminate the better the convergence rates will become, because finally, when all couplings between inner and outer unknowns are eliminated, the algorithm becomes a direct nested dissection solver again, and "converges" in one step. Of course we pay for this rapid convergence with the $O(N^{\frac{3}{2}})$ computing time for the eliminations and $O(N \log N)$ memory requirement due to the generated fill-in.

We therefore have to look for a compromise between eliminating enough couplings to achieve good convergence rates and keeping the number of eliminations small enough to maintain the $O(N)$ complexity with respect to computing time and memory requirement. A suitable heuristics for choosing the "correct" number of eliminated[1] unknowns can be found by analysing the underlying physical effects. Figure 5 shows a certain subdomain where a heat source on the left border is transported via a constant convection field towards the domain separator. Due to diffusion the former peak will extend over several mesh size steps after it has been transported to the separator. It is clear that using only one point on the separator is not enough to resolve the resulting heat profile (left picture).

[1] to "eliminate" an unknown in this context means that all couplings between "eliminated" unknowns are eliminated.

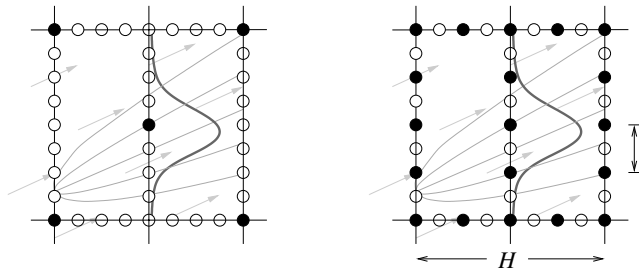


Fig. 5. Comparison of different elimination strategies: only the couplings between the black nodes are eliminated

Therefore, eliminating only the couplings between the black nodes of the left picture would not be sufficient to achieve good convergence. However, as we can see in the right picture, it is not necessary to eliminate all separator nodes like it is done in nested dissection.

The question is now how to choose the distance h between two eliminated nodes. From the underlying physics it is known that, for convection diffusion equations, the streamlines of the transported heat have a parabolic profile. Therefore, if the typical distance H between the unknowns grows by a factor of four, we can expect the heat profile to become twice as wide. Thus we can also double the distance h between the eliminated nodes. In other words, we choose $h \propto \sqrt{H}$ and the number of eliminated separator unknowns proportional to the square root of the total number of unknowns on the separator. In our algorithm, we implement this square root dependency by doubling the number of eliminated separator unknowns after every four bisection steps. This strategy is illustrated in figure 6.

The overall algorithm remains the same as in figure 3, but now the elimination matrices L and R no longer have only a small, fixed number of non-diagonal entries. If we have a subdomain with m separator unknowns, we eliminate the couplings between $O(\sqrt{m})$ inner unknowns and $O(\sqrt{m})$ outer unknowns which produces $O(m)$ entries in L and R. As a result of these extra eliminations, the

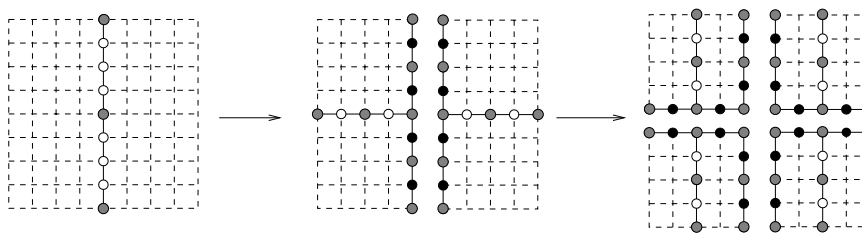


Fig. 6. Bisection with incomplete elimination, only the couplings between the grey nodes are eliminated

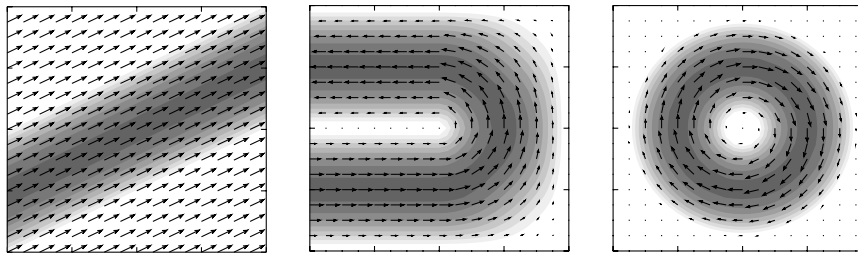
fill-in created in the system matrix K is now much bigger compared to the algorithms of section 2.2. Therefore, one can no longer afford to store the entire system matrix K for each subdomain as this would result in an $O(N \log N)$ memory requirement. However, if we choose the (weighted) Jacobi-method for relaxation, we only have to store the main diagonal of K as we can compute the residual separately (see steps U1-U3 of the algorithm in figure 3). This puts both the memory requirement and the number of operations needed for the set up of the system matrices back to $O(N)$. Of course the Jacobi-relaxation gives less satisfying convergence factors than the Gauss-Seidel-relaxation, but, as we will show in the next section, it is still possible to achieve reasonable performance.

4 Numerical Results

We tested our algorithm on the three different benchmark problems outlined in figure 7. Problem (a) indicates a flow with constant convection $a(x, y) = a$, problem (b) an entering flow that contains a 180 degree curve of the flow, and finally problem (c) a circular flow problem. Each problem was discretized on a square using standard second order finite difference discretization with a five point stencil.

As the elimination points are chosen independently of the flow field it is clear that the computation time and the memory used are independent of the type of the test problem. Figure 8 indicates that both computation time and memory requirement rise linearly with the number of unknowns. This can also be deduced by theoretical means.

Table 1 shows the convergence rates for test problem (c), but nearly identical results are obtained for test problems (a) and (b). Table 2 shows the average convergence rates for test problem (c) if the algorithm is used as a preconditioner for the BiCGStab method [7]. For the Poisson equation it is known that preconditioning with hierarchical bases leads to a logarithmic increase of the convergence rates, which corresponds well with the behaviour we observe in tables 1 and 2. We can see that the convergence rates are largely independent on the strength of convection as long as a certain ratio between convection strength and mesh



(a) straight convection (b) bent pipe convection (c) circular convection

Fig. 7. Convection fields of the three test problems

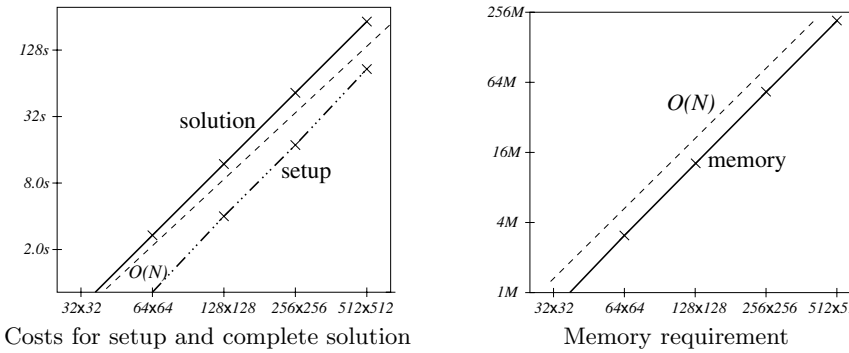


Fig. 8. Computation time and memory requirement

size is not exceeded. For stronger convection the heuristics of our elimination strategy no longer holds and convergence begins to break down.

	$a=0$	$a=1$	$a=2$	$a=4$	$a=8$	$a=16$	$a=32$	$a=64$	$a=128$	$a=256$
32×32	0.635	0.635	0.636	0.637	0.644	0.661	0.743	—	—	—
64×64	0.700	0.700	0.700	0.699	0.697	0.693	0.694	0.753	—	—
128×128	0.721	0.722	0.722	0.722	0.724	0.734	0.756	0.823	0.853	—
256×256	0.746	0.746	0.746	0.746	0.746	0.746	0.744	0.757	0.818	0.848
512×512	0.783	0.783	0.783	0.783	0.783	0.783	0.784	0.795	0.859	div

Table 1. Convergence rates for test problem c (circular convection), a denotes the strength of the convection. (Jacobi-relaxation with $\omega = \frac{2}{3}$)

	$a=0$	$a=1$	$a=2$	$a=4$	$a=8$	$a=16$	$a=32$	$a=64$	$a=128$	$a=256$
32×32	0.079	0.083	0.085	0.084	0.083	0.127	0.147	0.210	0.315	0.467
64×64	0.151	0.153	0.154	0.155	0.156	0.153	0.161	0.227	0.401	0.404
128×128	0.156	0.156	0.157	0.157	0.159	0.175	0.216	0.351	0.433	0.545
256×256	0.208	0.208	0.207	0.205	0.197	0.214	0.221	0.238	0.357	0.462
512×512	0.226	0.226	0.226	0.227	0.227	0.229	0.231	0.271	0.376	0.463

Table 2. Average convergence rates for test problem c (circular convection) using preconditioned BiCGStab and upwind discretization

A parallel implementation of our iterative nested dissection code was realized using MPI as message passing protocol. The distribution of the subdomains to the processors was done as was shown in figure 4. Figure 9 illustrates speedup and parallel efficiency of the parallel implementation for a problem with 512×512 unknowns on a cluster of SUN Ultra 60 workstations.

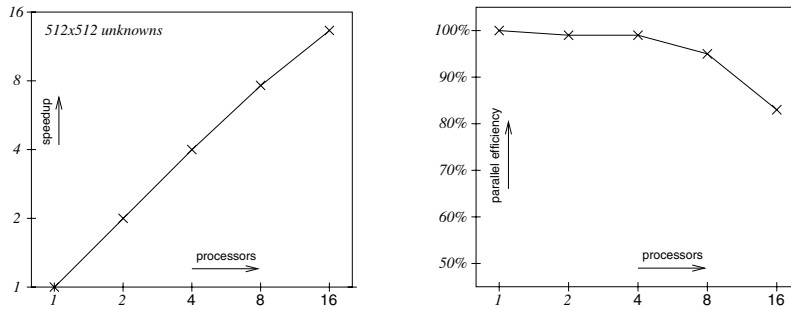


Fig. 9. Speedup and parallel efficiency on a cluster of SUN Ultra 60 Workstations

5 Present and Future Work

We are currently working on two topics which are still missing in the presented algorithm to become a suitable solver for practical convection type problems. The first topic is the treatment of complicated computational domains and of inner boundaries and obstacles, the other one is the removal of the $O(\log n)$ dependency in the convergence rates and to make them independent of the resolution of the mesh.

It seems that both topics can be successfully treated if we replace the hierarchical basis preconditioning by the usage of generating systems which turns the algorithm into a multigrid type method (see Griebel [8]). First numerical experiments indicate that the convergence rates seem to indeed become independent of mesh size, geometry and strength of convection (as long as the diffusion still dominates the convection on the finest mesh).

The efficient treatment of strongly convection dominated flow leads our list of future work. In the case of strong convection our elimination strategy is no longer appropriate as it depends on the existence of a certain amount of real diffusion. However, an efficient elimination strategy should be achievable either by eliminating couplings that are aligned with the direction of the flow or by choosing the eliminated couplings in an algebraic manner which leads to an AMG like method with fixed coarse grid selection.

References

1. Hüttl, R., Schneider, M.: Parallel Adaptive Numerical Simulation. Institut für Informatik, TU München, SFB-Bericht 342/01/94 A (1994)
2. Ebner, R.: Funktionale Programmierkonzepte für die verteilte numerische Simulation. PhD thesis, TU München (1999)
3. Ebner, R., Zenger, C.: A distributed functional framework for recursive finite element simulation. Parallel Computing **25** (1999) 813-826
4. George, A.: Nested Dissection of a Regular Finite Element Mesh. SIAM Journal on Numerical Analysis **10** (1973)

5. Ruge, J.W., Stüben, K.: Algebraic multigrid. In: McCormick, S.F. (ed.): Multigrid Methods. SIAM (1987) 73-130
6. Yserentant, H.: Hierarchical basis give conjugate gradient methods a multigrid type speed of convergence. Appl. Math. and Comput. **19** (1986) 347-358
7. van der Vorst, H.: Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Sci. Statist. Comput., **13** (1992) 631-644
8. Griebel, M.: Multilevel algorithms considered as iterative methods on semidefinite systems. SIAM Int. J. Sci. Stat. Comput., **15/3** (1994) 547-565

Low Communication Parallel Multigrid
A Fine Level Approach

Marcus Mohr

System Simulation Group of the Computer Science Department,
Friedrich-Alexander-University Erlangen-Nuremberg
Marcus.Mohr@cs.fau.de

Abstract. The most common technique for the parallelization of multigrid methods is grid partitioning. For such methods Brandt and Diskin have suggested the use of a variant of segmental refinement in order to reduce the amount of inter–processor communication. A parallel multigrid method with this technique avoids all communication on the finest grid levels. This article will examine some features of this class of algorithms as compared to standard parallel multigrid methods. In particular, the communication pattern will be analyzed in detail.

Keywords: elliptic pde, parallel multigrid, domain decomposition, communication cost

1 Introduction

There exists a great variety of different parallel architectures, ranging from clusters of workstations to massively parallel machines. In this paper we will consider the case that the number of processors is significantly smaller than the number of grid points and that memory is distributed. With this background the most common approach to the parallelization of numerical algorithms is grid partitioning. In the special case of multigrid methods nested levels of grids are employed. This approach introduces the need to communicate values related to the points on or near the inner boundaries. The cost of communication naturally limits the possible speedup of a parallel method. This cost can be alleviated by sophisticated programming techniques, e.g. by overlapping communication with calculation.

Generally the communication cost is determined by the number of messages that must be exchanged and the number of bytes that have to be transmitted. In multigrid the number of messages is proportional to the number of domains and therefore independent of the grid level. The number of bytes on the other hand is strongly related to the coarseness of the respective level since it is coupled to the number of interface points.

This leads to another aspect, namely the parallel efficiency of the algorithm, determined by the ratio between communication and computation. This ratio is directly proportional to the ratio of volume and surface of the subgrids and therefore becomes worse on the coarser grid levels. This is the starting point for

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 806–814, 2000.
© Springer-Verlag Berlin Heidelberg 2000

many approaches to improve parallel multigrid, like e.g. coarse grid agglomeration, multiple coarse grid and concurrent algorithms, see e.g. [6], or methods that employ different cycle schemes like e.g. the U–cycle [9].

A radical approach to reduce fine grid communication has been suggested by Brandt and Diskin in [3]. Here an algorithm that completely eliminates the need for inter–processor communication on several of the finest grids is presented. In the following we will examine a variant of this algorithm. We want to depict some of its characteristics and problems and compare its communication requirements to that of a "conventional" parallel multigrid algorithm. The algorithm will be introduced in Sect. 2. We will compare its communication requirements to that of a "conventional" parallel multigrid algorithm in Sect. 3 and analyze the communication pattern in Sect. 4.

2 Algorithm of Brandt & Diskin

In [3] Brandt and Diskin introduced a parallel multigrid algorithm completely without interprocessor communication on several of the finest levels. This was achieved by the use of segmental–refinement–type procedures, which were originally proposed as out–of–core techniques on sequential computers, cf. e.g. [1]. Since the bulk of communication takes place on the fine grids, it may be especially attractive to save this cost. The effectiveness of such a technique depends on specific machine and problem parameters. The potential benefits are most impressive when communication is slow and expensive with respect to computation.

Their algorithm can be described as follows. Starting from a hierarchy of grids, in a preliminary step all levels, except the coarsest one, are decomposed into as many overlapping subdomains as processors are available. We get sequences of nested subdomains, where a subdomain on a coarser grid occupies a larger area than the corresponding subdomain on the next finer grid. Each such sequence is then assigned to a processor, which starts on it a standard V–cycle, descending through its grid hierarchy. In this process it does not exchange data with its neighbours. When the second coarsest level is reached, the local (approximate) solutions from all processors are used to formulate a global coarse grid problem. This is then solved exactly by some unspecified algorithm, possibly of course a standard parallel multigrid method. The solution of the global coarsest grid problem is used by each processor to correct its local solution approximation on the second coarsest level. As in the following correction steps on the finer levels the values at the interfaces are included into the correction. After this step each processor finishes its V–cycle, again without communication with its neighbours. The algorithm uses the following basic principles to reduce communication:

- Clearly some exchange of information between the processors is inevitable to solve the problem. In the algorithm by Brandt and Diskin however, this data exchange is restricted to the coarse grid correction from the common coarsest grid. So there is no communication on the finer grid levels.

- To compensate errors introduced by the missing exchange of information, each subdomain has a buffer area around it, that fulfills two purposes. On the one hand, if an appropriate relaxation scheme, like e.g. red–black Gauß–Seidel is chosen, the buffer slows the propagation of errors due to wrong values at the interfaces. On the other hand, as in standard multigrid, the coarse grid correction introduces some high–frequency error components on the fine grid. Since the values at the interfaces cannot be smoothed, the algorithm cannot eliminate these components. But in elliptic problems high–frequency components decay quickly. So the buffer area keeps these errors from affecting the inner values too much.
- Nevertheless the algorithm will in general not be able to produce an exact solution of the discrete problem. While this may seem prohibitive at first glance, one should remember that, when solving a PDE, it is the continuous solution one is really interested in. Since the latter is represented by the discrete solution only up to the discretization error, the result of the algorithm is still valuable, as long as it can be guaranteed, that its algebraic error remains at least smaller than the discretization error.
- Since the overlap areas are included in the V–cycle, the algorithm trades communication for calculation. Thus savings in communication time may be partly compensated by the extra computation. This depends on several factors, such as size of the problem, number and arrangement of subgrids, extension of the overlap areas, and the MFLOP– and transfer–rates of the applied hard– and software. It will be further examined in the next section.

A crucial aspect of the above algorithm is of course the choice of an appropriate size for the overlap between subdomains. Until now there exists no general a priori error estimate that would allow to give a bound for the algebraic error depending on the overlap parameter J, but the experiments in [3,7,8] indicate that already small overlaps can produce reasonable results.

3 Efficiency Analysis

In this section we will explore how much overall computation time can be saved by the trade–off between communication and additional computation in the overlap areas. As mentioned above this depends on several architecture and problem parameters. To examine this question we compare a "standard" parallel V–cycle for the coarse grid correction (CGC) to an identical V–cycle that does not exchange data between processors, but instead employs overlapping subdomains. We model the times spent with computation and communication within the algorithm from the following simplifying assumptions:

- Data exchange between processors is performed by message passing.
- Communication and calculation are sequential and cannot be overlapped.
- A processor can send and receive messages simultaneously.

We now define a relative time saving T_{RS} per grid level in the following way

$$T_{\text{RS}} := \frac{T(\text{stand}) - T(\text{nocom})}{T(\text{stand})} \quad . \tag{1}$$

Here $T(\text{nocom})$ is the time for the variant without communication and $T(\text{stand})$ for the standard variant with communication. The times include all the work that has to be done for the specific grid level within one cycle of the CGC–scheme, that is smoothing, calculation of the defect, restriction of the defect, prolongation of the coarse grid solution and adding of the correction. They can be split in the following way

$$T(\text{stand}) = T_{\text{calc}}(\text{stand}) + T_{\text{comm}} \tag{2}$$

$$T(\text{nocom}) = T_{\text{calc}}(\text{stand}) + T_{\text{calc}}(\text{buffer}) \quad . \tag{3}$$

To determine the times for communication and calculation we use the following two models

$$T_{\text{calc}} = \frac{\gamma}{n_p}\Big(\nu F_{\text{smooth}} N_1 + F_{\text{multi}} N_2\Big) \tag{4}$$

$$T_{\text{comm}} = \frac{1}{n_p}\Big(\alpha\, M + \beta\, W\Big) \tag{5}$$

with the parameters

α	latency	F_{smooth}	# of FLOPs per point for smoothing
β	bandwidth	F_{multi}	# of FLOPs per point for CGC
γ	time per FLOP	N_1	# of points that are smoothed
n_p	# of processors	N_2	# of points included in the CGC
ν	sum of smoothing steps	M	# of messages to be exchanged
W	# of words (double precision values) that must be exchanged		

As model problem we consider a 2D elliptic boundary value problem discretized with a 5–point stencil on a logically rectangular grid. We assume the use of a 9–point stencil for restriction, bilinear interpolation as prolongation, and red–black Gauß–Seidel as smoother. For the 3D analogue, we use a 7–point discretization, a 27–point stencil for restriction, and trilinear interpolation. We assume that our global finest grid is quadratic/cubic with N^{dim} points ($N-1$ being a power of 2), and that we have a logical processor grid. Assuming furthermore that a non–overlapping domain decomposition approach is employed for the grid partitioning [5,8], we can, for given values of N, p_x, p_y and eventually p_z, and for a given overlap parameter J, derive the corresponding values of n_p, N_1, N_2, M, and W. We estimate the number of numerical operations per grid point as $F_{\text{smooth}} \approx 9\,(2\text{D})\,/\,11\,(3\text{D})$ and $F_{\text{multi}} \approx 7.5\,(2\text{D})\,/\,9\,(3\text{D})$.

Now we choose two different processor types, i.e. two different values for γ, one with 200 and the other one with 50 MFLOP. We compare the relative time saving T_{RS} per grid level for different grid sizes N and different communication

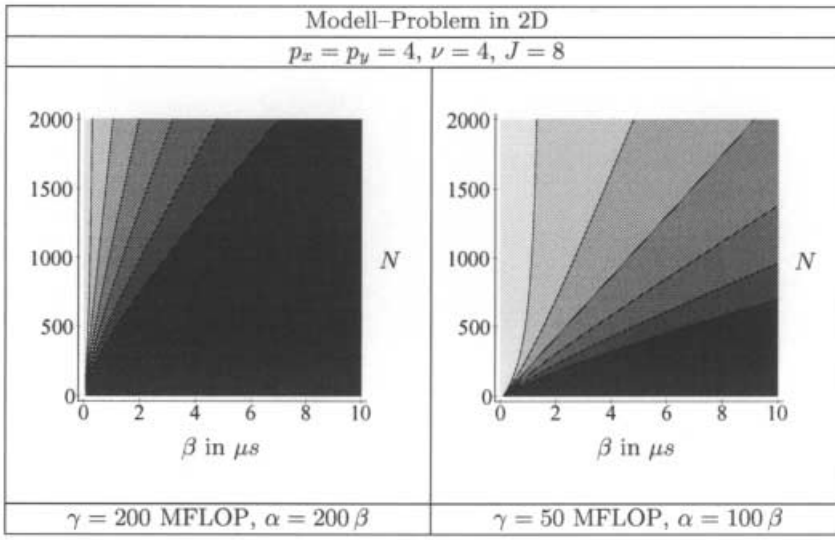


Fig. 1. Relative Time Saving per Grid Level for the 2D–Model Problem

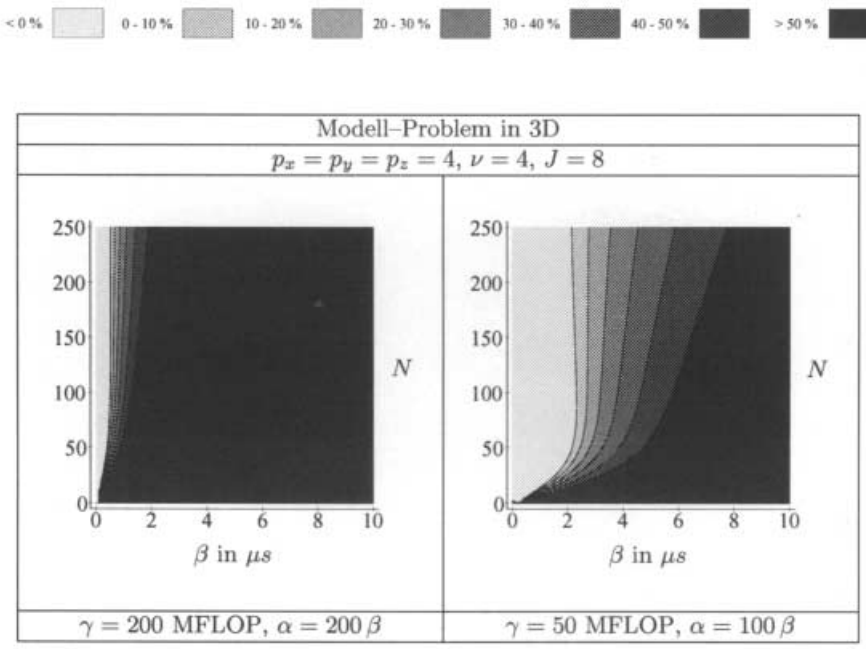


Fig. 2. Relative Time Saving per Grid Level for the 3D–Model Problem

speeds β. Latency effects are taken into account by assuming that $\alpha = \text{const} \cdot \beta$. The results are shown in Fig. 1 and 2.

Communication hardware may have widely varying performance characteristics. For orientation we present typical parameters for current implementations of the message passing interface (MPI). The following values have been taken from [4]:

	Myrinet	Fast Ethernet	Ethernet
α	$70\mu s$	$630\mu s$	$1150\mu s$
β	$0.36\mu s$	$2\mu s$	$11.4\mu s$
α/β	194	315	100

The results in Fig. 1 and 2 confirm the following properties of the approach. The value T_{RS} of the relative time saving grows with β. This was to be expected, since the more expensive communication, the greater the gain by replacing it with computation. However, as the grids become larger T_{RS} is reduced. Although the margin between communication time T_{comm} and additional calculation time T_{calc}(buffer) becomes more and more favorable as N grows, this is to some extent compensated by the growing influence of the regular computational work T_{calc}(stand) on the overall time. This can clearly be seen by comparing the figures for the 200 and the 50 MFLOP case. This compensation effect is significantly smaller in 3D than in 2D, since here the number of points and the amount of computation is of order $\mathcal{O}(N^3)$, whereas the number of interface points, responsible for communication, grows like N^2. In 2D, on the other hand, these values are N^2 and N, respectively.

As far as the remaining problem parameters are concerned, some properties can easily be derived. If we consider the overlap parameter J, it can be said that, as long as $N \gg J$, the additional computation grows almost linearly with J. So we have that $T_{\text{RS}} = C_1 - C_2 J$, with constants C_k depending on the other parameters. We found that the parameter ν had little influence on T_{RS}, but the number of processors n_p does. The relative time saving is more favorable, when there are more processors, because then there are fewer points on each subgrid and therefore the above mentioned compensation effect is reduced.

4 The Two Level Brandt–Diskin–Algorithm

In this section we will take a closer look at the two level version of the multilevel cycle in order to analyze in detail the communication pattern of the algorithm and show some of its properties. We do this by means of the model problem on the unit square Ω and restrict us to the case of two subdomains, which will suffice to explain the basic concept.

We discretize the problem on a fine grid $\Omega^h := \{x_{ij} \,|\, 0 \leqslant i, j \leqslant N\}$ and a coarse grid $\Omega^H := \{x_{2i2j} \,|\, 0 \leqslant i, j \leqslant N/2\}$. Here $x_{ij} = (ih, jh)$ denotes a grid point on the ith vertical and the jth horizontal line. We decompose the fine grid into two equally large parts $\Omega^{h,1}$ and $\Omega^{h,2}$ with the grid line $j = N/2$ as common

boundary. Now we augment each subgrid with a buffer zone of $J > 0$ grid lines and get the extended subdomains

$$\hat{\Omega}^{h,1} := \left\{ x_{ij} \;\middle|\; j \leqslant N/2 + J \right\} \quad \text{and} \quad \hat{\Omega}^{h,2} := \left\{ x_{ij} \;\middle|\; j \geqslant N/2 - J \right\} \quad . \tag{6}$$

The quantity $J > 0$ describes the amount of overlap between the two subdomains. It is required that J is a multiple of 2 to ensure that there are coarse grid points that coincide with interface points on the fine grid.

Each subgrid is now assigned to one processor p_1 and p_2. Starting from an approximation u_S^h to the discrete solution of the fine grid problem the algorithm is defined by iteratively performing the following seven steps:

1. Both processors perform separate pre–smoothing steps.
2. A global approximate solution is composed from the local solutions.
3. A global defect function is composed.
4. With the global solution and defect the right hand side of the coarse grid equation is calculated according to the FAS–scheme.
5. The coarse grid equation is solved exactly (by whatever means).
6. Each processor uses the coarse grid solution to correct its local approximate.
7. Both processors perform separate post–smoothing steps.

In step 1 and 7 the iteration method used is red–black Gauß–Seidel and there is no communication between the processors to update values along the interfaces. As a consequence the local solutions $u^{2,1}$ and $u^{2,2}$ will start to differ.

Steps 2 and 3 are of virtual character in the sense that no processor knows the global functions completely. Formally the global solution u^h is computed by taking the values of the local solutions in the interior of the subdomains $\Omega^{2,p}$ and their average along their common boundary. The same is done for the global defect function, so that we get

$$r^h(x_{i,j}) := \begin{cases} r^{h,1}(x_{i,j}) & \text{if } j < N/2 \\ r^{h,1}(x_{i,j})/2 + r^{h,2}(x_{i,j})/2 & \text{if } j = N/2 \\ r^{h,2}(x_{i,j}) & \text{if } j > N/2 \end{cases} \tag{7}$$

Here $r^{h,p} = f^h - L^h u^{h,p}$ denotes the local defect functions and f^h and L^h are to be interpreted as restrictions on the extended subdomains $\hat{\Omega}^{h,p}$. We point out that in general the so defined global defect function r^h is different from the defect $f^h - L^h u^h$ of the global solution function at points near the interface. We will return to this fact later on.

In a normal FAS–scheme the correction in step 6 would be performed as

$$u^h_{\text{new}} = u^h_{\text{old}} + I^h_H \left[u^H - I^H_h u^h_{\text{old}} \right] \tag{8}$$

where u^H is the solution of the coarse grid problem obtained in step 5, and I^h_H and I^H_h are a prolongation and a restriction operator. In the two level cycle the same is done, but since the global solution is not locally available, the local

solution is corrected instead. In order to understand how this correction leads to an update of information between the processors, we re-write the correction. We define a prolongation operator

$$J_H^{h,p} : \Omega^H \to \hat{\Omega}^{h,p} \tag{9}$$

calculating the values in $\hat{\Omega}^{h,p}$ from the respective values in $\Omega^H \cap \hat{\Omega}^{h,p}$ by bilinear interpolation and a restriction operator

$$J_{h,p}^H : \hat{\Omega}^{h,p} \to \Omega^H \tag{10}$$

which computes the values in $\Omega^H \cap \hat{\Omega}^{h,p}$ through injection and sets the values in $\Omega^H \backslash \hat{\Omega}^{h,p}$ to zero. Denoting by E^p the identity operator on $\hat{\Omega}^{h,p}$, we get

$$u_{\text{new}}^{h,p}(P) = \left(E^p - J_H^{h,p} J_{h,p}^H\right) u_{\text{old}}^{h,p}(P) + \left(J_H^{h,p} u^H\right)(P)\,. \tag{11}$$

Consider now a point $P = x_{ij}$ that lies in the overlap area $\hat{\Omega}^{h,1} \cap \hat{\Omega}^{h,2}$. For such a point both processors store a local solution value. Due to lack of communication during relaxation these values will in general differ prior to the correction step. The second term in the correction formula will lead to the same value on both processors, while the first term in (11 will vanish for points that can be found on the coarse grid. So in this case the values of the two processors at the point will coincide after the correction. For points not on the coarse grid, the first term will in general not vanish. This is due to interpolation errors. So the values of the two processors will be composed of a common part, namely $(J_H^{h,p} u^H)(x_{ij})$, and a disturbance, which is a remainder of the old function value at each processor. In this way an information update between the processors is embedded into the correction.

In the following, we want to point out some of the features of the two level cycle. First of all, since the algorithm represents an iteration on the pair of local solution functions $(u_i^{h,1}, u_i^{h,2})$, we have to analyze its convergence.

The experiments in [3] as well as the analysis for the special case of exact solvers as smoothers in [7] indicate that the algorithm will converge for every choice of initial values. But contrary to standard multigrid the result depends on the initial values. These can be grouped into equivalence classes according to their hierarchical offset, with every class converging to the same limit function. The reason for this is that the first term in (11) will always reproduce the hierarchical offset of the solution, while the second term represents a bilinear interpolation and therefore has no hierarchical offset at all. As a consequence the hierarchical offset of the initial values along the inner boundaries is never changed by the algorithm.

As another property we require that for a proper choice of the overlap parameter J the error with respect to the true discrete solution remains at least of the same order of magnitude as the discretization error. In [3,7] it was shown by experiment that this can already be achieved with an overlap of 2 to 4 grid lines. The accuracy can be improved, by a different way to compose a global

defect function r^h. As already mentioned, the global defect function when set up according to (7) does not match the defect $\bar{r}^h := f^h - L^h u^h$ of the global solution approximation. If we use $\bar{r}^h$ as the global defect, the algebraic error will be smaller than with the use of r^h and it will strongly be restricted to the vicinity of the common boundary of the subdomains. In this case also smaller overlaps are sufficient to keep the algebraic error smaller than the discretization error and this will often be achieved in a smaller number of cycles, [7,8].

This approach does not cause extra cost. Since the calculation of the defect $f^h - L^h u^h$ as well as the prolongation by full weighting and the application of the coarse grid operator L^H are linear operations the amount of communication needed to set up the coarse grid equation is the same independent of whether we use $\bar{r}^h$ or r^h.

5 Conclusions

In this paper we have shown that the approach by Brandt and Diskin to parallelize multigrid methods can be used to reduce overall computation time. This remains valid even for high–speed communication infrastructures as long as the processor speeds are fast enough. We have presented an analysis of the communication pattern of the two–level–version of their algorithm and have shown a way to improve its performance. Questions that remain open are a priori estimates for the algebraic error of the algorithm as well as convergence rates.

References

1. A. Brandt, Multi–level adaptive solutions to boundary value problems, Mathematics of Computation 31 (1977) 333 - 390.
2. A. Brandt, B. Diskin, Multigrid Solvers on Decomposed Domains, Contemporary Mathematics 157 (1994) 135 - 155.
3. B. Diskin, Multigrid Solvers on Decomposed Domains, M. Sc. Thesis, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, 1993.
4. M. Griebel and G. Zumbusch, Parnass: Porting gigabit–LAN components to a workstation cluster, in: W. Rehm, ed., Proceedings des 1. Workshop Cluster-Computing, 6.-7. November 1997, in Chemnitz, (Chemnitzer Informatik Berichte, CSR-97-05) 101-124.
5. M. Jung, On the parallelization of multi–grid methods using a non–overlapping domain decomposition data structure, Appl. numer. Math. 1 (1997) 119-138.
6. L. Matheson, R. Tarjan, Parallelism in Multigrid Methods: How much is too much?, Int. J. Parallel Programming 5 (1996) 397 - 432.
7. M. Mohr, Kommunikationsarme parallele Mehrgitteralgorithmen, Diplomarbeit, Institut für Mathematik, TU München, 1997.
8. M. Mohr, U. Rüde, Communication Reduced Parallel Multigrid: Analysis and Experiments, Technical Report No. 394, University of Augsburg, 1998.
9. D. Xie, L. Scott, The Parallel U–Cycle Multigrid Method, Virtual Proceedings of the 8th Copper Mountain Conference on Multigrid Methods, MGNET, 1997, (http://casper.cs.yale.edu/mgnet/www/mgnet.html).

Parallelizing an Unstructured Grid Generator with a Space-Filling Curve Approach

Jörn Behrens[1] and Jens Zimmermann[2]

[1] Munich University of Technology,
D-80290 München, Germany,
behrens@mathematik.tu-muenchen.de
www-m3.ma.tum.de/m3/behrens/,
[2] Ludwig-Maximilians-Universität,
D-80539 München, Germany,
jens.zimmermann@physik.uni-muenchen.de

Abstract. A new parallel partitioning algorithm for unstructured parallel grid generation is presented. This new approach is based on a space-filling curve. The space-filling curve's indices are calculated recursively and in parallel, thus leading to a very efficient and fast load distribution. The resulting partitions have good edge-cut and load balancing characteristics.

1 Introduction

Adaptive unstructured grid generation poses a nontrivial problem to parallelization approaches. To give an example: The adaptive simulation of atmospheric transport has been parallelized in [1]. However, grid generation has been used in its serial form, because at that time no adequate parallel grid generator was available. A new hierarchical adaptive grid generator (`amatos`: Adaptive mesh Generator for Atmospheric and Oceanic Simulations) has been implemented, based on modular software techniques. `pamatos` is the parallel implementation of `amatos`. `pamatos` can generate (i.e. refine, coarsen, and adapt corresponding to a given error criterion) dynamically changing grids in two space dimensions. The grid generator's behavior can be controlled by a simple programming interface (API). Parallelization is based on the message passing paradigm.

Besides the nontrivial problems associated to the data management in unstructured grid generation on parallel computers, one major problem is the mesh partitioning. Several authors have proposed different approaches, c.f. [4,7,11,12]. However, most of these methods require rather complicated calculations on the graph of the mesh.

Our approach follows a different line, as it is based on the geometry of the mesh. We propose a new construction scheme for a space-filling curve (SFC). Hilbert and Peano originally introduced SFCs for proving a set theoretical problem [6]. A SFC can be constructed such that it passes through all points of a multi-dimensional domain. This induces a mapping which allows to find a unique

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 815–823, 2000.
© Springer-Verlag Berlin Heidelberg 2000

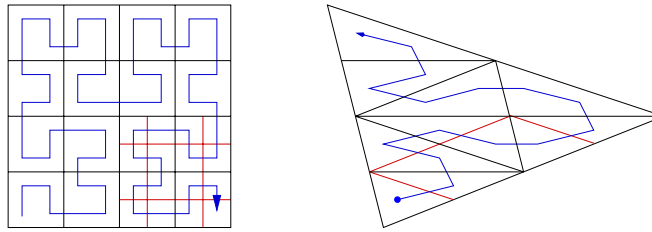


Fig. 1. A Hilbert-type space-filling curve for an adaptively refined rectangular grid (left). New space-filling curve in a locally refined triangular grid (left).

enumeration for all elements of the unstructured mesh. Consecutively numbered elements can then be distributed with almost optimal load balancing.

Roberts et al. [10] proposed several space-filling curve mechanisms for key generation in a hash table based mesh. Their bit-manipulation based algorithm also leads to a natural decomposition of the mesh. Griebel and Zumbusch [5] introduced space-filling curves to sparse grid adaptive methods. However, both approaches do not easily extend to non-rectangular domains.

In the following section we give a detailed description of our recursive algorithm for the SFC for non-rectangular domains. In section 3 we describe the grid generator which has been tailored for advection problems in atmospheric and oceanic simulation applications. Section 4 provides some numerical results and a conclusion is given in section 5.

2 Recursive Calculation of the Space-Filling Curve for Triangle Bisection

For two-dimensional rectangular domains Hilbert-type SFCs have the shape shown in Fig. 1. A recursive construction mechanism is given, for example, by Breinholt and Schierz [3]. There are a lot of different forms of space-filling curves for different purposes. Lafruit and Cornelis [9] use so called dove-tail space-filling curves for the parallelization of fast wavelet transforms.

The SFC is characterized by its corners. Each corner's index can be calculated recursively. For partitioning a mesh, a SFC has to be constructed such that each mesh cell holds at least one SFC corner. Each cell is indexed by the corresponding SFC index. Now, partitioning is easy: each processor receives an equally sized chunk of ascending indices.

Nearly optimal load balancing can be achieved. Griebel and Zumbusch [5] as well as Roberts et al. [10] showed that the partitions obtained by space-filling curve approaches are well behaved with respect to other measures (e.g. edge-cut). Our results in section 4 support this observation.

The algorithm for calculating a triangle index depends on three basic properties of our grid. First, the procedure mimics the bisection of triangles. Second,

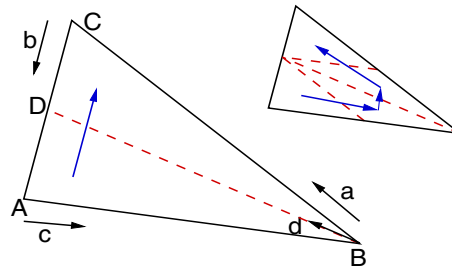


Fig. 2. Denotation of a triangle for the space-filling curve algorithm (see text).

the algorithm uses the center coordinates of each triangle as corners for the space-filling curve. Finally, in order to find new indices, a tabulated indexing scheme is used (see Table 1). There are three variables describing the triangle's state. Each triangle of a given macro triangulation has state 1 by definition.

Table 1. Table of states for the space-filling curve algorithm.

	state description variables			resulting states	
original state	bisection vector	direction of marked edge	direction of SFC segment	new state *i*	new state *ii*
0	d	b	+	5, add	3
1	d	b	-	2	4, add
2	a	c	+	7, add	1
3	a	c	-	0	6, add
4	c	a	+	1, add	6
5	c	a	-	7	0, add
6	b	d	+	3, add	4
7	b	d	-	5	2, add

To find the index for a refined triangle, we first have to describe the coarse triangle's state in detail. Figure 2 shows a triangle $\overline{ABC}$ in state 1, with an edge marked for bisection $\overline{CA}$. Vectors $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{d}$, denote different directions, used to describe the state. The triangle is divided by a line defined by $\boldsymbol{d}$ (dashed line in Fig. 2). The arrow inside the triangle indicates the direction of the SFC.

The algorithm for calculating the SFC index reads as follows (examples referring to Fig. 2 are given in brackets):

Algorithm 1 (Recursive space-filling curve)

1. *Determine original state from bisection, marked edge, and SFC direction (The SFC's direction is oriented from A to C, thus vector $\boldsymbol{b}$ with negative sign*

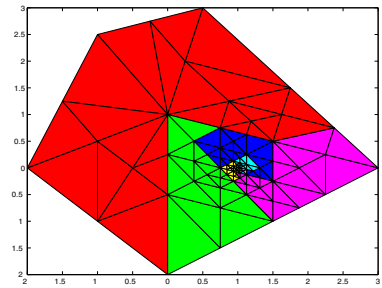


Fig. 3. Six partitions of a locally refined triangulation over a non-rectangular domain.

defines the direction of the SFC, and vector $\boldsymbol{d}$ *bisects the triangle. Therefore, triangle* $\overline{ABC}$ *is in state 1).*

2. *Determine position of refined triangle (triangle* $\overline{DBC}$*'s center is in the opposite direction as the marked edge's orientation).*
3. *Chose corresponding new state from Table 1 (we have new state ii for triangle* $\overline{DBC}$*).*
4. *Add offset to the index value, if required (as* $\overline{DBC}$*'s new state is 3 without the "add" attribute in the table nothing has to be done here).*

Table 1 contains all possible states. Applying Algorithm 1 recursively, yields a unique index for each triangle on the finest level of triangulation.

When using a hierarchical grid generator (as in our case), we start with a coarse macro triangulation. Utilizing Algorithm 1 for each triangle of the macro triangulation, adding an offset to the indices of each macro triangle, and supposing that the children of each macro triangle will not be deformed too much, this space-filling curve can be used for non-rectangular domains. An example is given in Fig. 3.

Each step of the following SFC partitioning algorithm can be executed in parallel. The recursive index calculation is embarrassingly parallel. Sorting can be done in parallel and moving elements is a parallel unstructured communication. To conclude, the whole SFC partitioning can be parallelized.

Algorithm 2 (SFC partitioning)

1. *Calculate SFC index for each triangle*
2. *Sort global index set*
3. *Cut global index set into chunks of equal size*
4. *Move all triangles which do not belong to the own chunk*

3 The Parallel Grid Generator

This section gives a brief description of the parallel mesh generator `pamatos`, using the new partitioning method. `pamatos` is a two-dimensional grid generator

for atmospheric and oceanic simulations. It implements a bisection refinement technique for the elements. `pamatos` is implemented in Fortran 90 utilizing a modular object oriented programming paradigm. A relatively small number of interface routines controls the behavior of the grid. There is a serial sister of `pamatos`, called `amatos`, that has been used in atmospheric simulations in [2]. In principle, the programming interface for the serial and the parallel mesh generator are similar, so the application programmer does not need to care about parallelization aspects. It is the task of `pamatos`, to distribute the work load evenly among the processors, to provide the necessary communication, etc.

The internal structure of `pamatos` is characterized by a hierarchy of different software layers. Communication primitives are built on top of a standard message passing library (at present MPI). The communication primitives layer hides the system dependent software parts away from the actual grid generator. On top of these, high level communication subroutines do the work required for movement and information update in an unstructured grid. High level communication subroutines are engaged both by the grid generation layer and by the list layer. The list layer facilitates programming in an unstructured grid generation process. An abstract procedure can be given as follows:

Algorithm 3 (Abstract list oriented procedure)

1. *Do the normal serial work until a data item is encountered that resides on a remote processor*
2. *Skip the part of work, depending on a non-available data item*
3. *Put the pointer to the required data item into the collection list and continue with serial work*
4. *If no serial work is left, communicate all data items in the collection list*
5. *resume work with (now available) data items*

With this mechanism, the methods developed for the serial grid generator have to be extended only by calls to the collection list and a communication step. This reduces the number of messages and increases the message length.

A typical adaptation step is given in Algorithm 4. Note that all the steps can be performed in parallel, however, there are several synchronization points required. Efficiency can be achieved only, if the grid is distributed evenly among the processors.

Algorithm 4 (Adaptation of the grid)

1. `[in parallel]` *Refine those elements, flagged for refinement*
2. `[synchronization point]` *Exchange edge information required for creation of an admissible triangulation*
3. `[in parallel]` *Coarse those elements, flagged for coarsening*
4. `[synchronization point]` *Exchange edge information required for creation of an admissible triangulation*
5. `[in parallel]` *Calculate new distribution*
6. `[synchronization point]` *Move data items to achieve load balancing, update connectivity information*

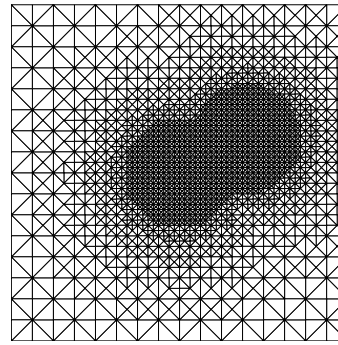


Fig. 4. Locally refined mesh, used in the test case described in the text. The mesh consists of approx. 4500 elements.

4 Numerical Examples

In order to show some numerical results a test case has been chosen that contains two regions of local refinement (see Fig. 4). Execution time and speedup for up to eight processors are given in Table 2. The computing environment consists of two SGI Origin 200 systems with four processors each (MIPS R10K, 225 MHz), connected by a dedicated Gigabit network connection.

Table 2. Execution times and relative speedup of the space-filling curve indexing.

no. of processors	2	4	8
time [ms]	68.7	26.5	14.9
speedup	1	2.6	4.6

Load balancing and relative edge-cut are given in Table 3. The *load balancing parameter* l is given by $l = e_{\max}/e_{\min}$, where $e_{\max}$ and $e_{\min}$ is the maximum and minimum number of elements on a single processor respectively. The *edge-cut* is the number of edges, that belong to more than one processor. This corresponds to the number of edges cut by a partition boundary in the dual graph (see [11]). The *relative edge-cut* is the edge-cut as a percentage of total edges.

The SFC partitioning scheme is intended for partitioning adaptive meshes in time-dependent simulations. Therefore, we compare results of the (serial) SFC scheme in an adaptive trace-gas simulation with Metis (version 4.0) [8]. Figure 5 gives load balancing and relative edge-cut for both schemes as a function of time. Metis's edge-cut is smaller than that of the SFC scheme. This should result in less communication in numerical calculations on the partitioned mesh. On the other

Table 3. Load balancing and relative edge-cut for the SFC-based algorithm on the model problem.

no. of processors	2	4	8
load balancing parameter	1.000	1.002	1.004
relative edge cut	0.03	0.04	0.06

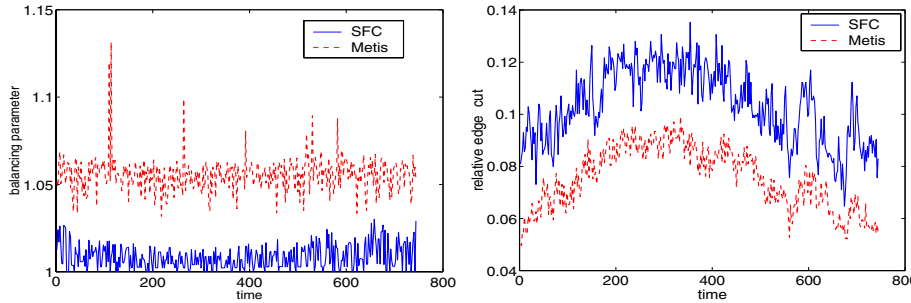


Fig. 5. Load balancing parameter (left) and relative edge-cut (right) for SFC and Metis in an adaptive time-dependent simulation.

hand, the SFC algorithm performs better with respect to the load balancing. This is not surprising, because (as noted before) optimal load balancing can always be achieved with SFCs.

Note that the SFC-based algorithm yields less data movement after remeshing. This is due to the geometry based distribution induced by the SFC, whereas Metis operates on the (dual) graph of the mesh and cannot take into consideration the location of each element. Figure 6 shows two consecutive meshes distributed with the SFC-based and the Metis algorithm respectively.

In Table 4 we compare computation times for a trace gas simulation on a mesh of 20,000 elements and 20026 edges. The given time for adaptation includes 12 adaptation steps and the partitioning in each step. Again, edge-cut and load balancing parameter is given. Note that the SFC-based algorithm is considerably faster.

Table 4. Load balancing, relative edge-cut, and timing of Metis versus SFC-based partitioning on the real-life problem.

	Metis	SFC-based
load balancing parameter	1.063	1.047
relative edge cut	0.026	0.042
time for adaptation [s]	16.5	6.5

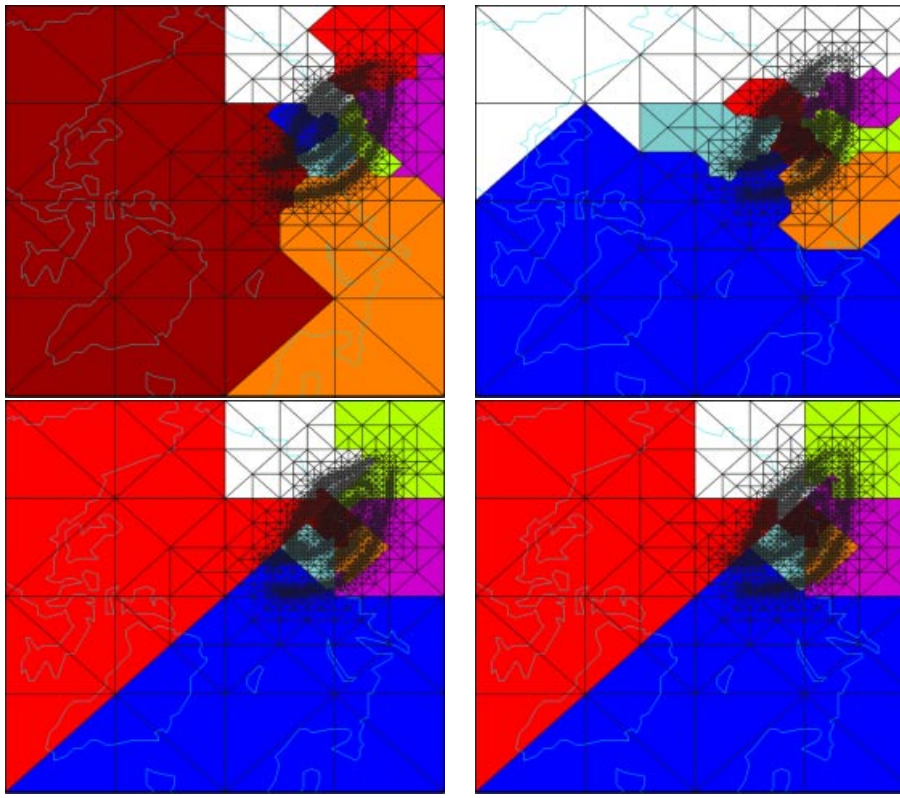


Fig. 6. Two consecutive meshes from a trace gas transport application. Metis redistributes almost every single element (upper row) while the SFC-based algorithm leaves most elements on the original processor (lower row).

5 Conclusions

In this article we have introduced a new recursive space-filling curve algorithm for the dynamic distribution of adaptively refined triangular meshes to many processors. The recursive indexing algorithm proves to be fast and easily parallelizable. The partitions resulting from our SFC have good load balancing and edge-cut characteristics. The SFC does not require rectangular domains. However, the SFC-based algorithm is not as versatile as graph partitioning algorithms, because it depends on the geometry of the mesh.

The proposed SFC algorithm is tailored for 2D bisection triangulations. It is supposedly easily extensible to 3D tetrahedral triangulations. Our algorithm is not suited for regular refinements yet.

We intend to use the new parallel grid generator `pamatos` in our atmospheric simulations. However, to be of practical use, a lot of work has still to be done. The results presented here are motivating to proceed further on this way.

References

1. J. Behrens. An adaptive semi-Lagrangian advection scheme and its parallelization. *Mon. Wea. Rev.*, 124(10):2386–2395, 1996.
2. J. Behrens, K. Dethloff, W. Hiller, and A. Rinke. Evolution of small-scale filaments in an adaptive advection model for idealized tracer transport. *Mon. Wea. Rev.*, 128, 2000. in press.
3. G. Breinholt and C. Schierz. Algorithm 781: Generating hilbert's space-filling curve by recursion. *AMS Trans. Math. Softw.*, 24:184–189, 1998.
4. R. Diekmann, D. Meyer, and B. Monien. Parallel decomposition of unstructured FEM-meshes. In *Proceedings of IRREGULAR 95*, volume 980 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, 1995.
5. M. Griebel and G. Zumbusch. Hash-storage techniques for adaptive mulitlevel solvers and their domain decomposition parallelization. *Contemporary Mathematics*, 218:279–286, 1998.
6. D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.*, 38:459–460, 1891.
7. M. T. Jones and P. E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 478–485. IEEE Computer Society Press, 1994.
8. G. Karypis and V. Kumar. *Metis – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minesota, Dept. of Computer Science/ Army HPC Research Center, Mineapolis, MN 55455, 1998. Version 4.0.
9. G. Lafruit and J. Cornelis. A space-filling curve image-scan for the parallelization of the two-dimensional fast wavelet transform. In *Proceedings of the 1995 IEEE Workshop on Nonlinear Signal and Image Processing*, http://poseidon.csd.auth.gr/Workshop/, 1995. Aristotle University of Thessaloniki, Aristotle University of Thessaloniki, Thessaloniki 540 06, P.O Box 451, Greece.
10. S. Roberts, S. Kalyanasundaram, M. Cardew-Hall, and W. Clarke. A key based parallel adaptive refinement technique for finite element methods. Technical report, Australian National University, Canberra, ACT 0200, Australia, 1997.
11. K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Par. Distr. Comp.*, 47:109–124, 1997.
12. C. Walshaw, M. Cross, M. Everett, and S. Johnson. A parallelizable algorithm for partitioning unstructured meshes. In A. Ferreira and J. D. P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, pages 25–46, Dordrecht, 1995. Kluwer Academic Publishers.

Solving Discrete-Time Periodic Riccati Equations on a Cluster*

Peter Benner[1], Rafael Mayo[2], Enrique S. Quintana-Ortí[2], and Vicente Hernández[3]

[1] Zentrum für Technomathematik, Fachbereich 3/Mathematik und Informatik, Universität Bremen, D-28334 Bremen, Germany;
benner@math.uni-bremen.de

[2] Departamento de Informática, Universidad Jaume I, 12080–Castellón, Spain;
{mayo,quintana}@inf.uji.es

[3] Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46071–Valencia, Spain;
vhernand@dsic.upv.es

Abstract. This paper analyzes the performance of a parallel solver for discrete-time periodic Riccati equations based on a sequence of orthogonal reordering transformations of the monodromy matrices associated with the equations. A coarse-grain parallel algorithm is investigated on a Myrinet cluster.

Key words: Discrete-time periodic Riccati equations, periodic linear control systems, parallel algorithms, multicomputers, cluster computing.

1 Introduction

Consider the discrete-time periodic Riccati equation (DPRE)

$$X_k = Q_k + A_k^T X_{k+1} A_k - A_k^T X_{k+1} B_k (R_k + B_k^T X_{k+1} B_k)^{-1} B_k^T X_{k+1} A_k, \quad (1)$$

where $A_k \in \mathbb{R}^{n\times n}$, $B_k \in \mathbb{R}^{n\times m}$, $C_k \in \mathbb{R}^{r\times n}$, $Q_k \in \mathbb{R}^{n\times n}$, $R_k \in \mathbb{R}^{m\times m}$, and p is the period of the system, i.e., $A_{k+p} = A_k$, $B_{k+p} = B_k$, $C_{k+p} = C_k$, $Q_{k+p} = Q_k$, and $R_{k+p} = R_k$. Under mild conditions, the periodic symmetric positive semidefinite solution $X_k = X_{k+p} \in \mathbb{R}^{n\times n}$ of (1) is unique [3]. DPREs arise, e.g., in the solution of the periodic linear-quadratic optimal control problem, model reduction of periodic linear systems, etc. [3].

Consider now the periodic symplectic matrix pencil, associated with the DPRE (1),

$$L_k - \lambda M_k = \begin{bmatrix} A_k & 0 \\ -Q_k & I_n \end{bmatrix} - \lambda \begin{bmatrix} I_n & B_k R_k^{-1} B_k^T \\ 0 & A_k^T \end{bmatrix} \equiv L_{k+p} - \lambda M_{k+p}, \quad (2)$$

* Supported by the Consellería de Cultura, Educación y Ciencia de la Generalidad Valenciana GV99-59-1-14, the DAAD Programme *Acciones Integradas Hispano-Alemanas*, and the Fundació Caixa-Castelló Bancaixa.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 824–828, 2000.
© Springer-Verlag Berlin Heidelberg 2000

where I_n denotes the identity matrix of order n. If all the A_k are invertible, the solution of the DPRE is given by the d-stable invariant subspace of the periodic monodromy matrices [5,8]

$$\Pi_k = M_{k+p-1}^{-1} L_{k+p-1} \cdots M_k^{-1} L_k, \quad \Pi_k = \Pi_{k+p}. \tag{3}$$

Note that the monodromy relation still holds if (some of) the A_k are singular and the algorithm presented here can still be applied in this case [3,5,8]. A numerically sound DPRE solver relies on an extension of the generalized Schur vectors method [5,8]. However, the parallel implementation of this QR-like algorithm renders an efficiency and scalability far from those of traditional matrix factorizations [4].

In this paper we follow a different approach, described in [2], for the solution of DPREs based on a reliable swapping of the matrix products in (3). In section 2 we briefly review the "swapping" method for solving DPREs and present a coarse-grain DPRE solver. A medium-grain parallel DPRE solver was investigated in [9]. In section 3 we report the performance of our approach on a cluster of Intel Pentium-II processors.

2 Parallel Solution of DPREs

In [2] an algorithm is introduced for solving DPREs without explicitly forming the monodromy matrices. The approach relies on the following lemma.

Lemma. Consider $Z, Y \in \mathbb{R}^{n\times n}$, with Y invertible, and let

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} Y \\ -Z \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{4}$$

be a QR factorization of $[Y^T, -Z^T]^T$; then $Q_{22}^{-1} Q_{21} = ZY^{-1}$.

By $(Y, Z) \leftarrow swap(Y, Z)$ we denote the application of the lemma to the matrix pair (Y, Z), where Y and Z are overwritten by Q_{22} and Q_{21}, respectively.

Applying the swapping procedure to (3), we obtain reordered monodromy matrices of the form

$$\Pi_k = \hat{M}_k^{-1} \hat{L}_k = (\bar{M}_k \cdots \bar{M}_{k+p-1})^{-1} (\bar{L}_{k+p-1} \cdots \bar{L}_k), \tag{5}$$

without computing any explicit inverse. The solution of the corresponding DPRE is then computed by solving the discrete-time algebraic Riccati equation (DARE) associated with the corresponding matrix pair $\hat{M}_k^{-1} \hat{L}_k$ [10].

The algorithm can be stated as follows [2]:

```
Input: p matrix pairs (L_k, M_k), k = 0, 1, ..., p-1
Output: Solution of the p DPREs associated with the matrix pairs
for k = 0, 1, ..., p-1
    Set L̂_k = L_k, M̂_{(k+1) mod p} = M_k
end
for t = 1, 2, ..., p-1
```

```
        for k = 0, 1, ..., p − 1
            (L_{(k+t) mod p}, M_k) ← swap(L_{(k+t) mod p}, M_k)
            L̂_k ← L_{(k+t) mod p} L̂_k
            M̂_{(k+t+1) mod p} ← M̂_{(k+t+1) mod p} M_k
        end
    end
    for k = 0, 1, ..., p − 1
        Solve the DARE associated with (L̂_k, M̂_k)
    end
```

The procedure is only composed of QR factorizations and matrix products. The computational cost of the reordering algorithm is $\mathcal{O}(p^2n^3)$ flops (floating-point arithmetic operations) and $\mathcal{O}(pn^2)$ for workspace. The cost of the solution of the p DAREs at the final stage is $\mathcal{O}(pn^3)$ flops and $\mathcal{O}(pn^2)$ for workspace.

Consider a parallel distributed-memory architecture, composed of n_p processors, P_0, P_1,..., P_{n_p-1}, and, for simplicity, assume that $p = n_p$. A coarse-grain parallel algorithm can be stated as follows:

```
    Input: p matrix pairs (L_k, M_k), k = 0, 1, ..., p − 1
    Output: Solution of the p DPREs associated with the matrix pairs
    for k = 0, 1, ..., p − 1 in parallel
        Assign (L_k, M_k) to processor P_k
        Set L̂_k = L_k , M̂_{(k+1) mod p} = M_k
        Send L_k to P_{(k+p−1) mod p}
        Receive L_{(k+1) mod p} from P_{(k+1) mod p}
    end
    for t = 1, 2, ..., p − 1
        for k = 0, 1, ..., p − 1 in parallel
            (L_{(k+t) mod p}, M_k) ← swap(L_{(k+t) mod p}, M_k)
            Send M̂_{(k+t) mod p} to P_{(k+p−1) mod p}
            L̂_k ← L_{(k+t) mod p} L̂_k
            Receive M̂_{(k+t+1) mod p} from P_{(k+1) mod p}
            Send L_{(k+t) mod p} to P_{(k+p−1) mod p}
            M̂_{(k+t+1) mod p} ← M̂_{(k+t+1) mod p} M_k
            Receive L_{(k+t+1) mod p} from P_{(k+1) mod p}
        end
    end
    for k = 0, 1, ..., p − 1 in parallel
        Solve the DARE associated with (L̂_k, M̂_k)
    end
```

This parallel algorithm only requires efficient serial kernels for the QR factorization and the matrix product, like those available in LAPACK and BLAS [1], and a serial DARE solver based, e.g., on the matrix disk function [2]. Moreover, the algorithm presents a highly regular and local communication pattern as, at each iteration of the outer loop t, the only communications neccesary are the left circular shifts of L_k and $\hat{M}_k$. Computation and communication are overlapped in the algorithm, and the computational load in the algorithm is perfectly balanced.

In case $p > n_p$, we can assign the matrix pairs (L_k, M_k) cyclically to the processors, and proceed in the same manner.

3 Experimental Results

All the experiments were performed on a cluster of Intel Pentium-II processors connected via a *Myrinet* switch, using IEEE double precision floating-point arithmetic ($\epsilon \approx 2.2 \times 10^{-16}$). BLAS and MPI implementations specially tuned for this architecture were employed [7]. Performance experiments with routine `DGEMM` achieved 200 Mflops (millions of flops per second) on one processor.

Figure 1 evaluates the efficiency of our coarse-grain parallel algorithm for n=100 and 200, and $n_p = p = 2, 4, \ldots, 10$ processors. We obtain efficiencies higher than 1 due to the better management of the memory achieved in the parallel algorithms, which have to deal with a smaller number of matrices. In this figure we also report the scalability of the parallel algorithm. For this purpose, we evaluate the Mflops per processor, with n^2p/n_p, $p = n_p$, fixed at 200. As the figure shows, the scalability is close to optimal.

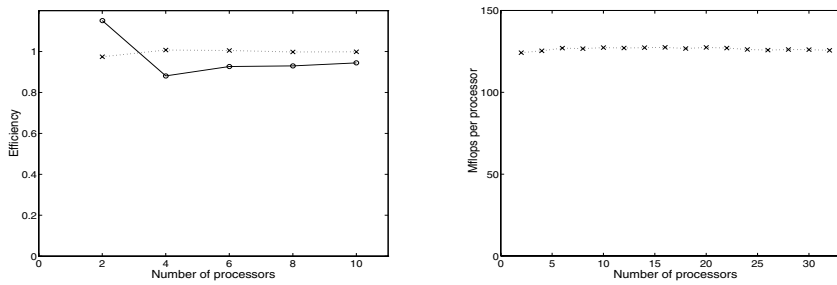


Fig. 1. Efficiency (left) and scalability (right) of the parallel algorithm for n=100 (solid line) and 200 (dotted line).

References

1. E. Anderson et al. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, 1994.
2. P. Benner. *Contributions to the Numerical Solution of Algebraic Riccati Equations and Related Eigenvalue Problems.* Dissertation, Fak. f. Mathematik, TU Chemnitz–Zwickau, Chemnitz, FRG, 1997.
3. S. Bittanti, P. Colaneri, and G. De Nicolao. The periodic Riccati equation. In S. Bittanti, A.J. Laub, and J.C. Willems, editors, *The Riccati Equation*, pp. 127–162. Springer-Verlag, Berlin, 1991.
4. L. S. Blackford et al. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.

5. A. Bojanczyk, G.H. Golub, and P. Van Dooren. The periodic Schur decomposition; algorithms and applications. In *Proc. SPIE Conference, 1770*, pp. 31–42, 1992.
6. G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, 1989.
7. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.
8. J.J. Hench and A.J. Laub. Numerical solution of the discrete-time periodic Riccati equation. *IEEE Trans. Automat. Control*, 39:1197–1210, 1994.
9. R. Mayo, E.S. Quintana-Ortí, E. Arias, and V. Hernández. *Parallel Solvers for Discrete-time Periodic Riccati Equations*. Lecture Notes in Computer Science. Springer–Verlag, 2000. To appear.
10. V. Mehrmann. *The Autonomous Linear Quadratic Control Problem, Theory and Numerical Solution*. Number 163 in Lecture Notes in Control and Information Sciences. Springer-Verlag, Heidelberg, July 1991.

A Parallel Optimization Scheme for Parameter Estimation in Motor Vehicle Dynamics

Torsten Butz[1], Oskar von Stryk[1], and Thieß-Magnus Wolter[2]

[1] Chair of Numerical Analysis (M2), Zentrum Mathematik,
Technische Universität München, D-80290 München, Germany
http://www-m2.ma.tum.de
[2] TESIS DYNAware, Implerstraße 26, D-81371 München, Germany
http://www.tesis.de

Abstract. For calibrating the vehicle model of a commercial vehicle dynamics program a parameter estimation tool has been developed which relies on observations obtained from driving tests. The associated nonlinear least-squares problem can be solved by means of mathematical optimization algorithms most of them making use of first-order derivative information. While the complexity of the investigated vehicle dynamics program only allows the objective gradients to be approximated by means of finite differences, this approach enables significant savings in computational time when performing the additionally required evaluations of the objective function in parallel. The employed low-cost parallel computing platform which consists of a heterogeneous PC cluster is well suited for the needs of the automotive suppliers and industries employing vehicle dynamics simulations.

1 Introduction

The numerical simulation of vehicle dynamics has gained considerable significance in automotive development, since it enables the thorough investigation of a novel vehicle in advance. Besides reducing the need for physical prototyping, real-time simulations may be used within hardware-in-the-loop test-benches which allow active control units, such as anti-lock braking systems and electronic stability programs, to be tested without danger for test driver and vehicle.

The development of complex electronic devices requires the virtual car to reproduce the behavior of the real vehicle in detail. Therefore, we employ a sophisticated vehicle model which comprises a suitable multibody system, including force elements and kinematical connections, as well as a realistic tire model. The use of a tailored modeling technique enables the entire vehicle dynamics to be described by a large system of ordinary differential equations.

Specifically for the use in a test-bench the calibration of the vehicle model need often be accomplished on the spot. For this purpose, nonlinear optimization algorithms and careful numerical differentiation can be combined to yield a parallel parameter estimation scheme which is suitable for low-cost computing platforms such as heterogeneous PC networks.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 829–834, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Simulation of Full Motor Vehicle Dynamics

The vehicle dynamics program *veDYNA* [1] which has been employed for the following investigations is developed and commercially distributed by TESIS DYNAware, München.

The vehicle model in *veDYNA* consists of a system of rigid bodies comprising the vehicle body, the axle suspensions and the wheels. In addition, partial models are employed to depict the characteristics of the drive train, the steering mechanism and the tires. The use of suitable minimum coordinates and generalized velocities avoids the need for algebraic constraints in the equations of motion [9]. Thus, the vehicle dynamics can fully be described by a system of 56 highly nonlinear ordinary differential equations. Due to the stiffness of the system its numerical integration is carried out by a semi-implicit Euler scheme.

For a realistic implementation of virtual test drives on the computer also models for the driver and the road have been developed [3].

The numerical results obtained from *veDYNA* show good agreement with real vehicle behavior. Simulations with time steps in the range of milliseconds may be carried out in real-time on reasonable PC hardware.

3 Estimation of Vehicle Parameters

The equations of motion for the vehicle model in *veDYNA* are summarized by

$$\dot{x}(t) = g\left(x(t), p, t\right) \tag{1}$$

with suitable initial values

$$x(t_0) = x_0 \ . \tag{2}$$

Here, $x(t) \in \mathbb{R}^{n_x}$ comprises the vehicle's state variables, and $p \in \mathbb{R}^{n_p}$ denotes the model parameters of interest which are constant for all times t. To adjust their values to the observed vehicle behavior, the nonlinear least-squares problem

$$\underset{p \in \mathbb{R}^{n_p}}{\text{minimize}}\ r(p) := \frac{1}{2}\left\|f(p)\right\|_2^2 := \frac{1}{2}\sum_{j=1}^{n_t}\sum_{i \in I_j}\left(\eta_{ij} - x_i(t_j, p)\right)^2 \tag{3}$$

must be solved. Here, η_{ij}, $i \in I_j$, are measurements of selected vehicle state variables at the times t_j throughout a driving test, and $x(t, p)$ denotes the corresponding numerical solution of (1), (2). Often additional box constraints

$$l_i \le p_i \le u_i, \quad i = 1, ..., n_p, \tag{4}$$

on the parameter range have to be considered which shall ensure optimization results compatible with the real vehicle properties.

For the solution of (3), (4) several gradient-based optimization methods as well as an evolutionary algorithm have been investigated [2]. In the sequel, we present results obtained from the Gauss-Newton method NLSCON [8], the Levenberg-Marquardt algorithm LMDER [7], the sequential quadratic programming method NLSSOL [5], and the implicit filtering code IFFCO [6], which is designed for solving noisy minimization problems.

4 Parallel Optimization

For the solution of the parameter estimation problem a program frame was implemented which integrates *veDYNA* in the course of the optimization [2]. Due to the complexity of the employed vehicle model and the closely coupled numerical integration, the required objective derivatives cannot be determined by automatic or internal numerical differentiation techniques, but have to be approximated by means of finite differences.

For the optimization with NLSCON, LMDER and NLSSOL the partial derivatives $\partial_i r(p) = f(p)^T \, \partial_i f(p)$ are obtained from the one-sided differences

$$(\partial_i f(p))_{\pm h_i} = \frac{f(p \pm h_i e_i) - f(p)}{\pm h_i} \tag{5}$$

depending on the feasibility of $p + h_i e_i$ or $p - h_i e_i$. Here, $e_i \in \mathbb{R}^{n_p}$ denotes the i-th canonical unit vector, and $h_i > 0$ is a finite difference increment which must be chosen carefully such as to account for truncation, condition, and rounding errors. The implicit filtering code IFFCO makes use of the central differences

$$(\partial_i r(p))_{2h_i} = \frac{r(p + h_i e_i) - r(p - h_i e_i)}{2h_i} \tag{6}$$

provided that both points are feasible; otherwise a one-sided difference is used as well. Accordingly, the computation of the gradient $\nabla r(p)$ requires n_p up to $2\, n_p$ additional evaluations of the objective function.

Since most effort is spent on the repeated integration of (1), (2), the computational time is much reduced by distributing these evaluations among further processors. For the one-sided differences the maximum speed-up is achieved, if n_p additional processors are available. In case of the centered differences one of the additionally required evaluations is performed by the client process, since the objective value at the current iterate need not be computed.

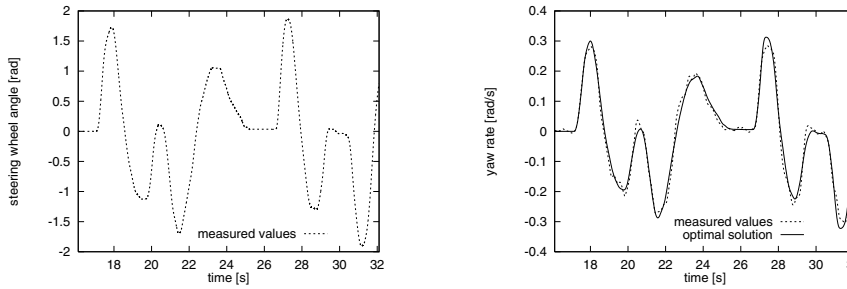
The communication between client and server processes across the network is handled by remote procedure calls. For this purpose, the ONC RPC library from Sun Microsystems, ported to Microsoft Windows, is used [4]. The exchange of data is done via the UDP transport protocol, since only arguments of moderate size are communicated.

5 Results

The above parameter estimation scheme was successfully employed to adjust the lateral vehicle dynamics properties in the *veDYNA* model of a passenger car [2]. Appropriate values for the remaining coefficients of the vehicle model had been validated by TESIS DYNAware beforehand.

The underlying data which was provided by an automotive supplier consisted of the steering wheel angle (cf. Fig. 1a) and the corresponding vehicle yaw rate recorded during multiple lane changes. The actual steering maneuver was preceded by a speed-up phase of 16.1 seconds. The sought vehicle parameters were

given by the x-coordinate of the center of gravity and the cornering stiffnesses at the front and rear wheels which determine the lateral tire forces.



(a) (b)

Fig. 1. Steering wheel angle (a) and vehicle yaw rate (b) for the lane change maneuver.

The optimization was carried out on a heterogeneous Windows NT 4.0 and Windows 98 network at TESIS DYNAware, München. Initial guesses

$$p^0 = (-1.242, 27075.5, 27075.5)^T \tag{7}$$

were chosen according to the default parameter values for the employed *veDYNA* vehicle model. The associated least-squares residual was $r(p^0) = 0.96115$.

The optimization produced a minimum residual $r(p^*) = 0.30324$ which was assumed for the parameter values

$$p^* = (-1.298, 16914.9, 15000.5)^T \tag{8}$$

computed by IFFCO. A comparison between the observed vehicle yaw rate and the corresponding simulation results for (8) is depicted in Fig. 1b. Good agreement is achieved between both characteristics. However, the small values of the estimated stiffnesses indicate that the kinematical axles of the vehicle model cannot depict the elastic properties of the actual suspension system exactly.

For reasons of comparison the numerical optimization was also carried out sequentially. In this case, the optimization including the computation of the gradients was done on a Dell 400 MHz PC where for each objective evaluation a CPU time of 8.1 seconds was needed.

In the parallel framework, the optimization was running on the same machine. The three function evaluations for the one-sided differences in NLSCON, LMDER and NLSSOL were performed on two Siemens 450 MHz PCs and a Dell 333 MHz notebook where the objective evaluations took 7.3 seconds and 8.8 seconds of CPU time respectively. The two additional evaluations required by IFFCO were carried out on a further Dell 333 MHz notebook and a Siemens 300 MHz PC. The corresponding CPU times were given by 8.8 and 9.7 seconds.

Table 1 shows a comparison between the results obtained from the different optimization codes [2]. The specified values consist of the least-squares residual at the respective optimal solution and the CPU times t_{seq} and t_{par} which were needed for the sequential and the parallel optimization. Their ratio, i. e., the achieved parallel speed-up, is given in the last column. Also listed is the number n_{seq} of objective evaluations during the entire optimization, and the share n_{par} that was performed by the client process in the parallel approach.

Table 1. Comparison of the computational results for the sequential and the parallel parameter estimation schemes.

Algorithm	$r(p^*)$	n_{seq}	t_{seq} [s]	n_{par}	t_{par} [s]	$t_{\text{seq}}/t_{\text{par}}$
IFFCO	**0.30324**	151	1223.6	64	543.1	2.25
LMDER	0.30341	52	**422.0**	25	207.8	2.03
NLSCON	0.30455	72	583.7	30	254.4	2.29
NLSSOL	0.30340	80	648.3	20	**185.3**	**3.50**

Applied to this problem the mentioned algorithms have produced meaningful parameter estimates with reasonably small residuals. The parallel execution of the finite difference computations reduces the required CPU times for all algorithms by more than a half. For NLSSOL, the computational time can be reduced to almost 25%, if equally fast remote processors are available. In case of the remaining optimization codes the achieved speed-up is significantly lower, since the employed line-search strategies do not allow a completely parallel treatment.

References

1. Anonymous: *veDYNA* User's Guide. TESIS DYNAware, München (1997)
2. Butz, T.: Parameter Identification in Vehicle Dynamics. Diploma Thesis, Zentrum Mathematik, Technische Universität München (1999)
3. Chucholowski, C., Vögel, M., von Stryk, O., Wolter, T.-M.: Real time simulation and online control for virtual test drives of cars. In: Bungartz, H.-J. et al. (eds.): High Performance Scientific and Engineering Computing. Lecture Notes in Computational Science and Engineering, Vol. 8. Springer-Verlag, Berlin (1999) 157–166
4. Gergeleit, M.: ONC RPC for Windows NT Homepage. World Wide Web, http://www.dcs.qmw.ac.uk/~williams/nisgina-current/src/rpc110/oncrpc.htm (1996)
5. Gill, P.E., Murray, W., Saunders, M.A., Wright, M.H.: User's Guide for NPSOL 5.0: A Fortran Package for Nonlinear Programming. Numerical Analysis Report 98-2, Department of Mathematics, University of California, San Diego (1998)
6. Gilmore, P.: IFFCO: Implicit Filtering for Constrained Optimization, User's Guide. Technical Report CRSC-TR93-7, Center for Research in Scientific Computation, North Carolina State University, Raleigh (1993)
7. Moré, J.J.: The Levenberg-Marquardt algorithm: implementation and theory. In: Dold, A., Eckmann, B. (eds.): Numerical Analysis. Lecture Notes in Mathematics, Vol. 630. Springer-Verlag, Berlin Heidelberg (1978) 105–116

8. Nowak, U., Weimann, L.: A family of Newton codes for systems of highly nonlinear equations. Technical Report TR 91-10, ZIB, Berlin (1991)
9. Rill, G.: Simulation von Kraftfahrzeugen. Vieweg, Braunschweig (1994)

Sliding-Window Compression on the Hypercube*

Charalampos Konstantopoulos, Andreas Svolos, and Christos Kaklamanis

Computer Engineering and Informatics Department, University of Patras and Computer Technology Institute, 11 Aktaiou and Poulopoulou, GR 118 51, Athens, Greece
{konstant,svolos,kakl}@cti.gr

Abstract. Dictionary compression belongs to the class of lossless compression methods and is mainly used for compressing text files [1, 2, 3]. In this paper, we present a parallel algorithm for one of these coding methods, namely the LZ77 coding algorithm also known as a sliding-window coding algorithm. Although there exist PRAM algorithms [4, 5] for various dictionary compression methods, their rather irregular structure has discouraged their implementation on practical interconnection networks such as the mesh and hypercube. However in the case of LZ77 coding, we show how to exploit the specific properties of the algorithm in order to achieve an efficient implementation on the hypercube.

1 Introduction

The main idea of the LZ77 algorithm [6] is that strings in the text are replaced with pointers to a previous occurrences of these strings in the text. Specifically, let $x[0 \cdots N-1]$ be the input string. Assume also the prefix $x[0 \cdots i-1]$ has been compressed so far. The dictionary at this moment consists of all the substrings $x[i-j \cdots k]$ where $j \in [1, \cdots, M]$, $k \in [i-j, \cdots, i-j+F-1]$ and M, F are two parameters of the algorithm. The next step is to find the longest prefix of $x[i \cdots N]$ which matches an entry of this dictionary. If this prefix is of length r and $x[i-q \cdots i-q+r-1]$ is the matching string in the dictionary ($q \in [1 \cdots M]$), then we replace the prefix $x[i \cdots i+r-1]$ with the pointer (q, r) and we proceed to the position $i+r$ of the input string. The string up to the position $i+r-1$ has now been compressed. Notice that if the character x_i does not occur within the last M preceding characters, then we cannot find any matching prefix at position i. In this case, we replace the character x_i with the pointer $(x_i, 1)$ [7].

In this paper, we present an efficient implementation of the LZ77 (sliding window coding) algorithm on the hypercube network. A basic assumption of our implementation is that the employed multiprocessor is fine grained with only limited local memory per processing element (PE). Taking advantage of the properties of sliding-window coding algorithms we show that this kind of algorithms can be efficiently implemented on the hypercube network by using only a small number of fast communication primitives.

* Work is supported in part by the General Secretariat of Research and Technology of Greece under Project ΠΕΝΕΔ 95 ΕΔ 1623.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 835–838, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 LZ77 Coding on the Hypercube

In the following, we present our parallel algorithm for sliding-window compression on a hypercube-based multiprocessor. The input string $x[0 \cdots N-1]$ is distributed one character per PE, i.e character x_i is initially stored in PE i where $i = 0, \cdots, N-1$. Our basic goal is to present an efficient parallel algorithm using the least amount of memory at each PE. For convenience, we also assume that $N = 2^n$.

In LZ77 coding algorithms the size M of the sliding window is finite and usually much smaller than the total length of the input string. In practice, a window of moderate size, typically $M \leq 8192$, can work well for a variety of texts [1]. Due to this small value, we can perform the required string matching operations using only simple search techniques without increasing the computational overhead unduly.

The parallel algorithm consists of two phases.

First phase. In this phase, for each position of the input string we find the longest substring starting at this position which also matches an entry in the adaptive dictionary. Specifically, for each position i of the input string we determine the longest common prefix of string $x[i \cdots i+F-1]$ with the strings $x[i-m_i \cdots i-m_i+F-1]$ where $m_i \in [1, \cdots M]$. This can be carried out in $O(M \log N)$ time by executing a series of M shift and prefix sum operations [8, 9]. If each PE can communicate with all its neighbors at the same time (all-port capability), the above complexity can be reduced to $O(M + \log N)$ time. This can be achieved by overlapping in time the execution of successive shift and prefix sum operations.

Second phase. It can be easily seen that the sequence of pointers obtained from the sequential LZ77 coding algorithm is a subset of the pointers derived from the first phase of the parallel algorithm. The goal in the second phase is to determine which of these pointers will be finally included in the compressed file.

This can be easily done in two stages by using the following simple technique. Let the pointer (m_i, l_i) of PE i point to the character $i + l_i$ of the input string, that is the first character after the longest common prefix at position i. The first pointer in the compressed file is definitely that of position 0, namely (m_0, l_0). The second pointer is that of position l_0, (m_{l_0}, l_{l_0}). The third pointer is that of position $l_0 + l_{l_0}$ and so on. Clearly, if we start from position 0 and follow the pointers defined above, we will eventually visit all the breakpoints[1] defined by the sequential LZ77 parsing. This sequential traversal of pointers can be performed in parallel in $O(\log N)$ steps by using the well-known pointer jumping technique [10].

As this is important, each PE saves the addresses of PEs it visits at each pointer jumping step. However, only $O(\log N)$ local memory at each PE is needed for this information, since each PE visits at most $\lceil \log N \rceil$ PEs during these steps.

We prove the following lemma:

[1] Breakpoints are the positions in the input text at which the parsing process splits the text.

Lemma 1. *If p_i, p_j are the pointers of PEs i, j respectively at a pointer jumping step , then $\forall$ i,j $i \leq j \Rightarrow p_i \leq p_j$.*

Proof. We prove the lemma by induction on the number of elapsing pointer jumping steps. In the first step, pointer p_i is equal to $i+l_i$ and thus the inequality $p_i \leq p_j$ can be written as $i + l_i \leq j + l_j$.

We distinguish two cases. If $i + l_i \leq j$, it is obvious that the above inequality holds. Now consider the case $i \leq j < i + l_i$. Clearly, character x_j belongs to the longest common prefix of position i and thus the longest common prefix at position j cannot be smaller than $i + l_i - j$ characters, that is $l_j \geq i + l_i - j \Rightarrow i + l_i \leq j + l_j$. Thus we have proved the lemma for the first step.

Assume now that the statement $\forall$ i,j $i \leq j \Rightarrow p_i \leq p_j$ holds for all the elapsing pointer jumping steps up to the step k. We will prove that it also holds for the step $k + 1$. Suppose that at step k position i points to position a_i and position a_i in turn points to position b_i. After step $k + 1$, position i will point to position b_i. We can prove that $\forall$ i,j $i \leq j \Rightarrow b_i \leq b_j$. If $j \geq b_i$, the proof is obvious. Consider now the case $i \leq j < b_i$. From the induction hypothesis, it follows that $a_i \leq a_j$. If $b_i > b_j$ then the statement $a_i \leq a_j \Rightarrow b_i \leq b_j$ does not hold, thereby contradicting the induction hypothesis. Thus the lemma is true for the step $k + 1$ as well. □

Now it is clear that each pointer jumping step can be performed using monotone routing [8, 9] in place of expensive sorting steps. Each step takes $O(\log N)$ time and thus the $O(\log N)$ pointer jumping steps of the first stage can be performed in $O(\log^2 N)$ time overall.

After the above pointer jumping steps, the next stage is to "mark" the positions of the input string that are breakpoints in the sequential LZ77 parsing. PE 0 knows that its position, position 0, is a breakpoint. It also knows $O(\log N)$ positions which are certainly breakpoints as well. These positions correspond to the PEs which it visited during the pointer jumping steps. Let $i_1, i_2, \cdots, i_{O(\log N)}$ be the addresses of these processors. PE 0 should notify them that hold breakpoints. After a PE, say PE i_k, receives the notification, it should in turn notify those PEs in the interval $[i_k+1 \cdots i_{k+1}-1]$ which it has visited during the pointer jumping steps. This process proceeds recursively and finally all the breakpoints of the sequential LZ77 parsing are marked. It can be easily seen that this recursive marking can be carried out by reversing the steps of the first stage. At each step, each PE that has already received a notification packet sends such a packet to the PE that it had visited at the corresponding pointer jumping step of the first stage. The communication complexity of each step is only $O(\log N)$ since we can use monotone routing again.

We have described the second phase of the parallel LZ77 coding algorithm. The complexity $O(\log^2 N)$ of this phase is mainly due to the fact that pointer jumping steps are performed along a string of length N. However, it is possible to limit pointer jumping steps along shorter segments of the input string, thereby largely decreasing the complexity of the second phase. Let us see how this can be done. After the end of the first phase, each PE i holds the pointer (m_i, l_i) in

its local memory. Then, using a prefix sum operation, each PE i estimates the expression $max_i = max(e_0, e_1, \cdots, e_{i-1})$[2] where $e_i = l_i + i$ ($O(\log N)$ delay). One can easily notice that if $i \geq max_i$, pointer (m_i, l_i) is definitely included in the compressed file. The corresponding position i is called cut-point; cut-point is a position in the input string for which we are certain in advance that it is one of the breakpoints of the LZ77 parsing [1]. Clearly, cut-points split the input string into non-overlapping substrings and thus we can execute the second phase of the parallel LZ77 coding algorithm independently for each substring. Due to the shorter length of these substrings, the complexity of the second phase is largely decreased. Specifically, if L is the length of the longest substring between two successive cut-points, the second phase can be executed now in $O(\lceil \log L \rceil \cdot \log N)$ time. In practice, $L \simeq 100$ since cut-points almost always occur well under 100 characters apart [1].

3 Conclusions

We presented an efficient parallel algorithm for LZ77 coding on the hypercube network. General simulations of PRAM dictionary compression algorithms on the hypercube surely leads to solutions with high communication overhead. However, by carefully examining the way LZ77 parsing splits the text into phrases, we managed to considerably lower the communication overhead. In doing so, we used only a small set of communication primitives which can be efficiently executed on the hypercube network. In addition, we further enhanced the performance by exploiting known facts from the text compression (frequent cut-points).

References

[1] T. C. Bell, J. G. Cleary, I. H. Witten, Text Compression Prentice Hall Advanced Reference Series Computer Science, (1990).
[2] K. Sayood, Introduction to Data Compression Morgan Kaufmann Publishers Inc. (1996).
[3] J. A. Storer, Data Compression Methods and Theory, Computer Science Press, Rockville, MD (1988).
[4] L. M. Stauffer, D. S. Hirschberg, Dictionary Compression on the PRAM, Parallel Processing Letters 7 (3) 1997 297–308.
[5] M. Farach, S. Muthukrishnan, Optimal Parallel Dictionary Matching and Compression (Extended Abstract), in Proc. SPAA 1995, 244–253.
[6] J. Ziv, A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Trans. Inf. Theory 23 (3) 1977 337–343.
[7] T. C. Bell, Better OPM/L Text Compression, IEEE Trans. Communications COM-34 (12) 1986 1176–1182.
[8] S. Ranka, S. Sahni, Hypercube Algorithms with Applications to Image Processing and Pattern Recognition, Springer Verlag (1990).
[9] T. F. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers, San Mateo, CA (1992).
[10] J. Jàjà, An Introduction to Parallel Algorithms, Addison-Wesley (1992).

[2] We assume $max_0 = 0$.

A Parallel Implementation of a Potential Reduction Algorithm for Box-Constrained Quadratic Programming*

Marco D'Apuzzo[1], Marina Marino[2], Panos M. Pardalos[3], and Gerardo Toraldo[2]

[1] Seconda Univerità di Napoli & CPS-CNR, Napoli, Italia,
dapuzzo@matna2.dma.unina.it
[2] Univerità di Napoli Federico II & CPS-CNR, Napoli, Italia,
(marino, toraldo)@matna2.dma.unina.it
[3] University of Florida, Gainesville Florida, USA,
pardalos@cao.ise.ufl.edu

Abstract. In this paper we describe a parallel version of the potential reduction algorithm for MIMD distributed memory machines, in which the computational kernels arising at each step of the algorithm are concurrently performed by using standard parallel software environments. This approach is shown to be very effective, in contrast to what happens in the active set strategies where the linear algebra computational kernels represent a serious drawback to an effective parallel implementation. The computational results show the effectiveness of our approach.

1 Introduction

In this paper we describe a parallel implementation of an *interior point* algorithm, proposed by Han et al. [8] for the *Box-Constrained Quadratic Programming (BCQP)* problem, based on the parallelization of the linear algebra kernels arising at each iteration. Our problem can be stated as follows:

$$\text{minimize} \qquad f(x) = \frac{1}{2}x^T Ax - b^T x \tag{1}$$

$$\text{subject to} \quad x \in \Omega = \{x : l \leq x \leq u\}. \tag{2}$$

Here A is a $n \times n$ symmetric, positive definite matrix, b, l and u are known n-vectors. Moreover, all inequalities involving vectors are interpreted componentwise, and $\nabla f(x) = Ax - b$ is the gradient of f. For the sake of simplicity we assume that the bounds, l and u, are finite.

Until the advent of interior point algorithms in the mid 1980s [9], the computational scene was dominated by the *active set* algorithms, implemented in many mathematical software libraries (NAG, IMSL). In contrast with active set

* This work was partially supported by the MURST national projects "Analisi Numerica: Metodi e Software Matematico" and "Algorithms for Complex Systems Optimization".

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 839–848, 2000.
© Springer-Verlag Berlin Heidelberg 2000

algorithms which generate a sequence of extreme points, a generic interior point method generates a sequence of points in the interior of Ω.

The algorithm we consider in the present paper is the so-called *potential reduction* algorithm [8,10,14], successfully used for linear complementarity and bound constrained quadratic problems. We consider the case in which the matrix A is dense and a direct solver is used in the inner iterations. Then, each iteration of the potential reduction algorithm has $O(n^3)$ computational complexity, and therefore much higher than in an active set strategy where the computational cost per iteration varies between $O(n^3)$ (if the Cholesky factorization must be recomputed) and $O(nm)$ (if either a projected gradient step or a simple factorization update must be computed). On the other hand, it has been observed that in practice the number of iterations in interior point algorithms does not depend on the number of variables; in contrast the number of iterations for an active set strategy strongly depends on the number of variables (see for example the discussion and the computational results in [11]).

We note that the computational kernels of the two approaches are quite different, and so are their parallel features. The parallel implementation of the projected gradient method proposed by Moré and Toraldo [11] has been recently discussed by D'Apuzzo et al. [5,6], who pointed out that the main drawback in implementing an active set strategy are the linear algebra operations that at each iteration must be performed on matrices and vectors reduced with respect to the current set of the free variables. In theory, working just on the free variables represents an advantage, because if few variables are free at the solution, eventually very small problems must be solved. Unfortunatly, the standard linear algebra package *LAPACK* [1] can not be used, as well as the *BLAS*, [1] to work on submatrices and on subvectors. In addition, the very poor parallel properties of the factorization updating process (due to the large amount of communication among processors), joint with the possible unbalanced data distribution (due to the modification of the active set at each iteration) represent a serious drawback for an efficient parallel implementation. Linear algebra operations on matrices and vectors whose size changes dynamically at each iteration and drastically reduces in the last iterations, are non-suitable for a good parallel implementation. This is also confirmed from the fact that these non standard operations are not implemented in the most popular high performance mathematical libraries. Because of this, the parallel performances of the projected gradient method in [6] appear to be not very satisfactory, despite the considerable efforts required by the algorithm implementation. We note that the key issues arising in the development of a parallel version of the algorithm in [11] are common to any nonlinear optimization algorithms based on active set strategies (matrix updating processes, linesearches, operations with subvectors, etc.).

On the other hand, the computationally expensive iterations of the interior point algorithms turn out to be an advantage, since they can be efficiently parallelized. In this paper we describe a parallel version of the potential reduction algorithm for MIMD distributed memory machines, in which the computational kernels arising at each step of the algorithm are concurrently performed by using the standard *ScaLAPACK* [3] environment.

The outline of our paper is as follows. In §2, we review the potential reduction algorithm as presented in [8]. The main issues arising in making a parallel implementation of the potential reduction algorithm are described in §3. Finally, in §4 we present the computational results of the implementation of the parallel algorithm on a MIMD distributed memory machine. Moreover, a comparison among the sequential versions of the projected gradient [11] and the potential reduction algorithms is shown.

2 The Potential Reduction Algorithm for Quadratic Problems with Box Constraints

In this section we outline the primal-dual Potential Reduction (PR) algorithm for solving the BCQP problem (1-2), that was proposed by Han et al. in [8], who extended the primal-dual PR algorithm for the convex linear complementarity problem, developed by Todd and Ye in [12].

Problem (1)-(2) can be transformed to the standard form

$$\text{minimize} \qquad f(x) = \frac{1}{2}x^T Ax - b^T x \tag{3}$$

$$\text{subject to} \qquad x + z = e \;\text{ and }\; x, z \geq 0, \tag{4}$$

where $z \in \mathbb{R}^n$, e is the unit vector, and whose dual is:

$$\max\; e^T y - \frac{1}{2}x^T Ax, \qquad \text{s.t.}\;\; s_x = Ax - b - y \geq 0 \;\text{ and }\; s_z = -y \geq 0, \tag{5}$$

where $y \in \mathbb{R}^n$. The duality gap of (3)-(4) and (5) is given by:

$$\Delta = \frac{1}{2}x^T Ax - b^T x - (e^T y - \frac{1}{2}x^T Ax) = x^T s_x + z^T s_z \geq 0. \tag{6}$$

The interior points x, z, s_x, and s_z are characterized by using the primal-dual potential function of Todd and Ye [12]:

$$\phi(x, z, s_x, s_z) = \rho \ln(x^T s_x + z^T s_z) - \sum_{i=1}^{n} \ln(x_i(s_x)_i) - \sum_{i=1}^{n} \ln(z_i(s_z)_i),$$

where $\rho \geq 2n + \sqrt{2n}$. The relationship between the duality gap and the potential reduction function was shown by Ye [14], who proved that

$$\text{if } \phi(x, z, s_x, s_z) \leq -O((\rho - 2n)L) \text{ then } \Delta < 2^{-O(L)},$$

where L is the size of input data (see [14]). Therefore, the method is based on the minimization of the potential function, in order to minimize the duality gap. Kojima et al. [10] showed that the Newton direction of the nonlinear system

$$\begin{cases} XS_x e = \frac{\Delta}{\rho} e \\ ZS_z e = \frac{\Delta}{\rho} e, \end{cases} \tag{7}$$

with fixed Δ, guarantees a constant reduction in the potential function. In (7) the upper-case letter (X) designates the diagonal matrix of the vector (x) in lower-case, and the slack vectors are

$$z = e - x, \qquad s_x = Ax - b - y, \qquad s_z = -y. \tag{8}$$

Following the Newton direction, given the kth iterates x^k and y^k, that are interior feasible points for the primal (3)-(4) and the dual (5), it can be verified that the search direction $\{\delta x, \delta z, \delta y\}$ can be computed by solving the systems:

$$\left(\bar{D}A\bar{D} + S_x^k Z^k + S_z^k X^k\right)\left(\bar{D}^{-1}\delta x\right) = \frac{\Delta^k}{\rho}\left(\bar{D}(X^k)^{-1}e - \bar{D}(Z^k)^{-1}e\right) - \bar{D}(s_x^k - s_z^k) \tag{9}$$

$$-\bar{D}\delta y = S_z^k X^k\left(\bar{D}^{-1}\delta x\right) + \left(\frac{\Delta^k}{\rho}\bar{D}(Z^k)^{-1}e - \bar{D}s_z^k\right), \tag{10}$$

where $\delta x = -\delta z$. In (9) and (10) $\bar{D}$ is the diagonal matrix, used to deal with ill-conditioning [2,13], whose diagonal coefficients are given by:

$$\bar{d}_{ii} = \sqrt{x_i^k z_i^k} \quad \text{for } i = 1, \cdots, n.$$

After obtaining the direction, the step length $\bar{\theta}$ can be simply chosen as

$$\begin{aligned}\bar{\theta} = \beta \max\{\theta : x^k + \theta\delta x \geq 0, e - (x^k + \theta\delta x) \geq 0, -y^k - \theta\delta y \geq 0,\\ \text{and } {s_x}^k + \theta(Q\delta x - \delta y) \geq 0\} \quad \text{for some } 0 < \beta < 1,\end{aligned} \tag{11}$$

and the $(k+1)$th iterates are generated as

$$x^{k+1} = x^k + \bar{\theta}\delta x\ , \quad y^{k+1} = y^k + \bar{\theta}\delta y.$$

Han et al. in [8] observed that for ρ large enough x^{k+1} and y^{k+1} guarantee a constant reduction in the potential function. This suggests that the use of the line-search is unnecessary and can be replaced by (11), which is used in our implementation. Now we are ready to summarize the primal-dual PR algorithm:

PR algorithm for BCQP

`Choose` $x^0 = .5e$ `and` $\alpha = \max\{1, 3 \parallel Ax^0 - b \parallel - \frac{e^T}{2n}(Ax^0 - b)\}$
$z^0 = e - x^0$; $y^0 = -\alpha e$; ${s_x}^0 = Ax^0 - b - y^0$; ${s_z}^0 = -y^0$
$\Delta^0 = (x^0)^T {s_x}^0 + (z^0)^T {s_z}^0$; $k = 0$
while $(\Delta^k \geq 2^{-L})$ **do**
  begin
    `Compute` δx `and` δy `from` (9) and (10)
    `Let` $\bar{\theta}$ `be given by (11)`
    $x^{k+1} = x^k + \bar{\theta}\delta x$
    $z^{k+1} = e - x^{k+1}$
    $y^{k+1} = y^k + \bar{\theta}\delta y$
    ${s_x}^{k+1} = Ax^{k+1} - b - y^{k+1}$

$s_z{}^{k+1} = -y^{k+1}$
$\Delta^{k+1} = (x^{k+1})^T s_x{}^{k+1} + (z^{k+1})^T s_z{}^{k+1}$
$k = k + 1$
end
endwhile

Regarding the theoretical complexity of the algorithm, the following theorem holds [8].

Theorem 1. *Let assume A and b have integer entries; let $\rho \geq 2n + \sqrt{2n}$ and $L = 2n^2 + [\log |P|]$ where P is the product of the nonzero integer coefficients appearing in A and b. Then, the algorithm terminates in $O((\rho - 2n)L)$ iterations and each iteration uses $O(n^3)$ arithmetic operations.*

3 A Parallel Version of PR Algorithm

In this section we present a parallel version of the algorithm described in the previous section to solve the BCQP problem. Our target computational environment is the *MIMD distributed memory* one.

The basic idea to obtain such parallel version was to perform concurrently the linear algebra operations at each iteration of PR algorithm. Therefore, we focused our attention on the choice of appropriate strategies for an efficient parallel implementation of these computational kernels. A first step in such direction was to consider and to use the ScaLAPACK environment.

ScaLAPACK [3] is a well known high performance library designed to solve linear algebra problems on distributed memory multiprocessors. It is based on a parallel computational model which provides a convenient and flexible message passing framework for developing efficient and portable parallel software without worrying about the underlying architecture details. Such a model is implemented into BLACS [7], a message passing library that provides tools to perform basic communication operations on matrices or submatrices. We assume that the computational model consists of a two dimensional grid of processes, where each process stores and operates on blocks of block-partitioned matrices or vectors. The blocks are distributed in a cyclic block fashion over the processes and their size represents both the distribution block size and the computational blocking factor used by processes to perform most of the computation. In particular the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy.

The choice of the block size has a major impact on load balance, communication cost, and loop and index computation overhead of the parallel algorithm, determining its performance and scalability. A larger block size results in a greater load imbalance, but reduces the frequency of the communication among processes. On the other hand, a smaller block size leads to a better load balance, but results in an inefficient use of the memory hierarchy. However, the

performance of the ScaLAPACK routines is not very sensitive to the block size as long as the extreme cases are avoided [3].

In developing our parallel algorithm, we assumed to have a process grid with R process rows and C process columns and each process is identified by its coordinates within the grid using the notation r, c, $0 \leq r < R$, $0 \leq c < C$. We decomposed the Hessian matrix A of BCQP problem into square blocks $A_{i,j}$, $i, j = 1, ..., NB$, where $NB = N/BS$ and BS is the block size, in order to obtain a square block matrix of dimension NB. These blocks are uniformly distributed among the process grid so that the process $P_{r,c}$ holds the blocks $A_{i,j}$ such that $(i-1) mod R = r$ and $(j-1) mod C = c$. We also decomposed vectors into NB blocks, each of dimension BS, and distributed them along each process column according to the matrix distribution.

Next, we analyze the PR algorithm. Each iteration requires:

- 2 inner products (to compute Δ),
- 2 matrix-vector products (to compute $\bar{\theta}$ and s_x),
- a linear system solution (to compute δx).

The above linear algebra operations can be efficiently implemented using suitable routines from the ScaLAPACK library. In this context, starting from the described data distribution strategy, our goal was to allow processes to perform parallel computation and communication, so that each of them holds only the blocks of the resulting vectors needed to update the corresponding blocks of the iterates x^k, y^k. Specifically, analyzing a single iteration of the algorithm, the basic computational and communication steps are organized as follows:

- `Computation of` δx.
  It is performed in two steps. The first one consists of the concurrent generation of the linear system (9) to be solved. Each process generates appropriate blocks of the coefficient matrix and known vector, so that the linear system is distributed among the process grid according to the initial distribution strategy. The second step consists of the parallel block solution of the linear system, and the final vector solution is distributed along the process columns as well as the other vectors.
- `Computation of` δy.
  The coefficient matrix of the linear system (10) related to δy is diagonal, so each process can compute explicitly its blocks of the vector solution.
- `Computation of` θ.
  It requires to find the maximum value (11) among values for which some vectors are non negative. One of such vectors is obtained by performing a matrix-vector product. Once this computation is done in parallel, each process computes the maximum value related to the components of the vectors it holds. After that, a global combine operation among all processes is needed to obtain each process has the global maximum.
- `Vectors updating`.
  Each process is ready to update vectors x, y, z, and s_z, while to update s_x another parallel matrix-vector product is needed.

– `Computation of` Δ
  The computation of (6) requires two parallel inner products, then two global combine operations.

4 Computational Results

In this section we present the results of an implementation of the parallel PR algorithm on an *IBM RS6000 SP* machine hosted by CPS-CNR.

The IBM SP is a MIMD distributed memory machine with 16 Power2 Super Chip Thin nodes running at 160 Mhz, each with 512 MBytes of memory and a 128 KBytes L1 cache. The nodes are interconnected via a proprietary switch, with a peak bi-directional bandwidth of 110 MBytes/sec.

All implementations were carried out using the AIX XL Fortran compiler and the BLACS, which are implemented on the top of the IBM proprietary version 2.3 of MPI message passing library. Furthermore some routines from *PBLAS* and ScaLAPACK were used. The PBLAS [4] is a package representing the parallel version of the BLAS and it is addressed to distributed basic linear algebra operations with the aim of simplifying the parallelization of linear algebra codes. The PBLAS is used to develop the ScaLAPACK library and it is part of it. Specifically we used the `PDSYMV` and `PDDOT` routines from PBLAS to perform matrix-vector and inner products respectively; the `PDPOSV` routine from ScaLAPACK for solving a linear system with symmetric and positive definite matrix; the `DGSUM2D` and `DGAMN2D` routines from BLACS to perform the combine communication operations required by the parallel algorithm.

In our computational experiments we used test problems similar to the problems introduced by Moré and Toraldo in [11], and also used in the paper by Han et al. [8]. In order to construct such test problems, 5 parameters are considered: the number of variables (n), the magnitude of the condition number of A ($ncond$), the number of active constraints at the solution x^* ($na(x^*)$), the magnitude of the amount of degeneracy of x^* ($ndeg$) [1], and the number of active constraints at the starting point x^0 ($na(x^0)$). For the results in this section we set $ncond = 6$ and $ndeg = -6$, that is we consider problems that are both midly degenerate and ill-conditioned. Furthemore, the value of $na(x^*)$ is set to 50, which means that about half of the constraints are active at the solution.

In our experiments we first performed some computations in order to compare the PR algorithm with a method based on an active set strategy. In particular, we considered the algorithm (henceforth called PG algorithm) proposed by Moré and Toraldo, that combines standard active set strategies with the gradient projection method. Specifically, this algorithm alternates gradient projection steps, in order to identify the optimal active set, and Newton steps, in order to explore the face determinated by such active set. We used an existing Fortran 77 code based on PG algorithm, that we compiled and executed on one processor of the IBM SP.

[1] The degeneracy parameter $ndeg$ is such that $[\nabla f(x^*)]_i = \pm 10^{r_i ndeg} \quad \forall i : x_i^* = u_i$ or $x_i^* = l_i$, where $r_i \in (0,1)$ is randomly generated.

In the PR algorithm the starting point should be an interior point of the feasible domain, so we do not consider the parameter $na(x^0)$, and we choose x^0 as the middle point of the feasible domain. Furthemore, since according to some computational results Han et al in [8] concluded that, from the convergence rate point of view, a good combination of values for the two parameters β and ρ is given by $\beta = 0.99$ and $\rho = n^{1.5}$, we use such values in our experiments. For the stopping criteria, we used $\Delta^k \leq 10^{-5}$, and additionally $\Delta^k/(1+|f(x^k)|) \leq 10^{-8}$.

For the PG algorithm the choice of the active set at the starting point is made using the same technique used for the solution, on the basis of the parameter $na(x^0)$. In particular, the values of $na(x^0)$ are restricted to 10, 50 and 90, that is we considered only the cases when about 10%, 50% and 90% of the constraints are active at the starting point, respectively. Iterations are stopped when $\|g(x^k)\| \leq 10^{-10}$, where $g(x^k)$ is the residual of the Kuhn-Tucker conditions.

Table 1 shows the CPU times (in seconds) and the number of iterations required to solve the test problem with both PR and PG algorithms on one processor of the IBM SP2, varying the problem size from 504 to 2520. For the PG algorithm we report times for the 3 mentioned different percentages of active constraints at starting point and their average. In the last column of the table ratios between the CPU times of PR algorithm and the average times of PG algorithm are shown. We first observe that these results confirm that for the PR algorithm the number of iterations is independent of the problem size. This is not true for the PG algorithm, where the number of iterations varies when the problem size grows up. Such behavior justifies the fact that, although the computational cost of each iteration of PR algorithm is greater than that of PG algorithm of about 1 order of magnitude, the PR algorithm always requires less time than the PG algorithm. So we can state that PR algorithm is competitive with a well representative algorithm from the active set class. However it is worth to note that we are comparing results obtained from an implementation of PR algorithm that uses collections of high-performance linear algebra routines tailored and optimized to the IBM SP architecture with results obtained from an existing PG code which has not been tailored to the target machine. In any case, the results obtained show that the PR algorithm can be considered an effective tool to solve medium size dense problems.

Next, we investigated the efficiency of the parallel version of PR algorithm. The number of processors used is 6, 9 and 12, logically organized as 2×3, 3×3, and 3×4 grids, respectively. In Table 2 we report the overall CPU times (T_p), the CPU times (t_p) needed to solve the linear systems (9), and the efficiency values (E_p) obtained varying the problem size from 1008 to 3528. Note that the CPU times T_p and t_p are given in seconds and refer to the parallel algorithm on p processors, and E_p is defined as $E_p = T_1/pT_p$. In the last column of the table, the efficiency values for 12 processors with respect to 6 processors are shown. Moreover, the results shown have been obtained using $BS = 42$. Our computational experiences done using different block size values showed that $BS = 42$ leads to good performance on the considered target machine. Moreover, such value allows to have uniform data distribution among processes for

Table 1. CPU times and number of iterations of PG and PR algorithms

n	PG 10%		PG 50%		PG 90%		PG Av. PG		PR		PR/PG
	Itrs	Time	Itrs	Time	Itrs	Time	Itrs	Time	Itrs	Time	
504	229	10.2	195	7.6	197	7.9	207	8.6	15	3.9	0.45
756	210	23	216	25	219	27	215	25	16	12	0.48
1008	265	75	246	60	248	60	253	65	16	25	0.39
1512	321	200	257	147	256	153	278	167	16	74	0.45
2016	278	304	233	225	243	254	248	261	16	170	0.65
2520	326	548	344	595	309	509	326	551	17	328	0.59

the considered problem sizes. The experimental results have to be considered satisfactory and confirm that PR algorithm has been parallelized successfully, obtaining good parallel performances even with small/medium size dense problems. We observe that efficiency values greater than 0.5 were obtained for 6 and 9 processors with problems of size greater than 2000, and for 12 processors when $n \approx 2500$. Finally, we point out that, as expected, the most contribution to the total amount of work is given by the solution of the linear systems, which takes about the 90% of the overall time for $n > 2000$, as the values of t_p show. We conclude that the parallel performance of PR algorithm is heavly influenced by the performance of the ScaLAPACK routine used to solve the linear systems.

Table 2. CPU times and efficiency of the parallel PR algorithm

n	T_1 (t_1)	T_6 (t_6)	T_9 (t_9)	T_{12} ($t_1 2$)	E_6	E_9	E_{12}	$E_{6/12}$
1008	25 (21)	9.4 (7.4)	9.6 (5.9)	9.1 (6.6)	0.44	0.29	0.23	0.52
1512	74 (64)	22 (19)	18 (14)	18 (15)	0.56	0.46	0.34	0.61
2016	170 (153)	42 (38)	33 (28)	31 (29)	0.67	0.57	0.46	0.68
2520	328 (300)	78 (71)	57 (51)	54 (49)	0.70	0.64	0.51	0.72
3024		114 (108)	83 (78)	77 (73)				0.74
3528		184 (178)	134 (122)	118 (113)				0.78

5 Concluding Remarks

We presented the results concerning the development and the implementation of a parallel PR algorithm for the BCQP problem based on the parallelization of the computational linear algebra kernels arising in the algorithm. A comparison with a projected gradient method showed that PR algorithm is competitive with active set strategies. Moreover, our approach to the parallelization of the

algorithm seems to be very effective. In conclusion, we believe that interior point algorithms represent a very appealing approach for the development of efficient parallel software for the more general nonlinear optimization problem.

Acknowledgments
The authors would like to thank the anonymous referees for their helpful suggestions.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarlig, A. McKenney, S. Ostrouchov and D. Sorensen - *LAPACK User's Guide, Second Edition* - SIAM, Philadelphia, PA (1995).
2. J.L. Barlow and G. Toraldo - *The effect of diagonal scaling on projected gradient methods for bound constrained quadratic programming problems* - Opt. Methods and Soft., vol. 5 n.3, pp. 235-245 (1995).
3. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley - *ScaLAPACK User's Guide* - SIAM, Philadelphia, PA (1997).
4. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley - *A Proposal for a Set of Parallel Basic Linear Algebra Subprograms* - LAPACK Working Note #100, University of Tennessee (1995).
5. M. D'Apuzzo, V. De Simone, M. Marino, and G. Toraldo - *Parallel Computational Issues for Box-Constrained Quadratic Programming* - Ricerca Operativa, vol. 27 n.81/82, pp. 125-141 (1997).
6. M. D'Apuzzo, V. De Simone, M. Marino, and G. Toraldo - *Modifying the Cholesky Factorization on MIMD Distributed Memory Machines* - in "High Performance Algorithms and Software in Nonlinear Optimization", R. De Leone et al. (eds.), Kluwer Academic Publishers, pp. 125-141 (1998).
7. J. Dongarra and R.C. Whaley - *A User's Guide to the BLACS v1.0* - Technical Report UT CS-95-281, LAPACK Working Note #94, University of Tennessee (1995).
8. C.G. Han, P.M. Pardalos, and Y. Ye - *Computational aspects of an interior point algorithm for quadratic problems with box constraints* - in "Large-Scale Numerical Optimization", T. Coleman and Y. Li eds., SIAM, PA, pp. 92-112 (1990).
9. N. Karmarkar - *A New Polynomial Time Algorithm for Linear Programming* - Combinatorica, vol. 4, pp. 373-395 (1984).
10. M. Kojima, N. Megiddo, and A. Yoshise - *An $O(\sqrt{n}L)$ Iteration Potential Reduction Algorithm for Linear Complementarity Problems* - Mathematical Programming, vol. 50, pp. 331-342 (1991).
11. J.J. Moré and G. Toraldo - *Algorithms for bound constrained quadratic programming problems* - Numerische Mathematik, n. 55, pp. 377-400 (1989).
12. M.J. Todd and Y. Ye - *A Centered Projective Algorithm for Linear Programming* - Mathematics of Oper. Res., vol. 15, pp. 508-529 (1990).
13. A. van der Sluis - *Condition numbers and equilibration of matrices* - Numerische Mathematik, vol. 14, pp.14-23 (1969).
14. Y. Ye - *An $O(n^3L)$ potential reduction algorithm for linear programming* - Mathematical Programming, vol. 50, pp. 239-258 (1991).

Topic 12
European Projects

Roland Wismüller and Renato Campo

Topic Chairmen

Between 1994 and 1998 projects in the area of High Performance Computing and Networking (HPCN) were funded by the European Commission in the framework of the ESPRIT Programme. The three papers in this session originate from three different ESPRIT projects. Since its original conception in the early 1980s the European R&D Programme in Information Technologies (ESPRIT) evolved from the initial technology-push supply-side approach towards IT applications and technology take-up with an increasing involvement of the users. Currently as part of the V Framework Programme the IST Programme continues its support of research, development and demonstration of Information Society Technologies.

The first paper presents work carried out in the NEPHEW project, which promotes cluster computing as a cost effective platform for parallel computing. In particular, the project investigates clusters consisting of PCs with Windows NT and an SCI interconnect. Three computationally intensive applications are implemented on this platform: Restoration of historic film material, low-cost flight simulation for pilot training, and reconstruction of medical PET (positron emission tomography) images. In order to reduce the effort of implementing and maintaining parallel applications, the PeakWare programming environment has been ported to the target platform. The paper by W. Karl et al. reports on first experiences with this environment gained with the PET application. The experiences are promising, both with respect to the ease-of-use of the programming environment and the communication speed of the PC cluster.

The next two papers both are devoted to the development of high performance simulators, which reflects the focus of the ESPRIT programme on applications in the area of simulation. Partially by chance and partially because it is an important topic for our society, they are both in the area of traffic control.

The SEEDS project especially addresses the problem of ground traffic control at airports. It developed a simulation environment for the analysis and evaluation of distributed traffic control systems. The distributed environment consists of components for, e.g., scenario generation, visualization, actor modeling and automatic decision support systems, which are connected via a CORBA middleware. The paper by T. Hruz et al. presents one special component of this environment: The Airport Management Database System. It presents the requirements and the design of this component and also some interesting experiences with the industrial use of Java, CORBA and Open Software tools.

The HIPERTRANS project puts a focus on the modeling and simulation of urban road networks. The traffic simulator is intended to be used for the testing and the assessment of traffic control systems, for the training of operators, and

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 849–850, 2000.
© Springer-Verlag Berlin Heidelberg 2000

for forecasting, i.e. for predicting the effects of control decisions. Obviously, the latter use requires that the traffic simulation can be performed faster than the real traffic evolves. Thus, high performance simulation techniques are needed. The paper authored by S. E. Ijaha et al. presents an overview on the simulation system, which uses parallel execution based on model partitioning to achieve the performance necessary for forecasting.

NEPHEW: Applying a Toolset for the Efficient Deployment of a Medical Image Application on SCI–Based Clusters

Wolfgang Karl[1], Martin Schulz[1], Martin Völk[2], and Sibylle Ziegler[2]

[1] Lehrstuhl für Rechnertechnik und Rechnerorganisation, LRR–TUM
Institut für Informatik, Technische Universität München, Germany
{karlw, schulzm, voelk}@in.tum.de
[2] Nuklearmedizinische Klinik und Poliklinik, MRI-NM,
Klinikum Rechts der Isar, Technische Universität München, Germany
s.ziegler@lrz.tum.de

Abstract. With the rise of cluster architectures, high–performance parallel computing is available to more users than ever before. The programming of such systems, however, has not yet improved beyond the cumbersome programming style used in message passing libraries like PVM and MPI. In order to open clusters to a broader audience, higher level programming environments have to be designed with the goal of giving the end user an easier access to parallel computing. Such an environment is currently being developed within the NEPHEW project allowing the graphical specification of global dependencies. This work presents the NEPHEW approach and discusses its applicability using an example application from the area of nuclear medical imaging, the reconstruction of PET (Positron Emission Tomography) images.

1 Motivation

With the rise of commodity clusters interconnected with high-speed system area networks (SANs), such as SCI [6][3] or GigaNet [8], high performance parallel computing has become available to more users than ever before. The programming of such systems, however, is still mostly based on pure message passing in the form of libraries like PVM [2] and MPI [7]. This is generally perceived as inappropriate for most end users prohibiting cluster architectures to be fully exploited. The NEPHEW[1] (*NE*twork of *P*Cs *HE*terogeneous *W*indows-NT Engineering Toolset) project attempts to tackle this problem by providing a high level programming environment for the implementation of parallel applications on top of commodity Windows NT based clusters. To ease the implementation process, lower the learning curve, and therefore giving also non computer scientist better

[1] The NEPHEW project is funded by the European Commission in the Fourth Framework Programme at the ESPRIT Project EP29907 / NEPHEW. More information is available under http://wwwbode.in.tum.de/Par/arch/smile/nephew/.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 851–860, 2000.
© Springer-Verlag Berlin Heidelberg 2000

access to cluster architectures, the deployed tool from Matra Systems & Information, called PeakWare [10], offers a graphical development environment that allows a simple and intuitive abstraction of communication paths within the application. Only the actual application functionality has to be hand coded in conventional sequential style using C. In addition, the tool environment provides mechanisms allowing for arbitrary mappings between functions and nodes they are supposed to be executed on.

PeakWare then automatically combines these individual parts into the final application. Any functionality needed for global operations and communication are generated by PeakWare from the graphical description and combined with the sequential routines. The generated final binary can be executed on a Windows-NT based cluster according to the specified node mappings.

An additional advantage of this approach lies in the fact that the PeakWare toolset is by default independent of the underlying communication network and is generally intended to be used with standard Fast Ethernet. For applications with higher communication demands, however, other interconnection technologies have to be deployed. Within the NEPHEW project, the Scalable Coherent Interface (SCI) [3] is used for this purpose. SCI is a state-of-the-art IEEE standardized [6] high performance SAN. It provides a end-to-end bandwidth of up 118 MB/s and process-to-process latencies as low as 2 μs. This high performance is offered to the user through PeakWare in two different ways: using a transparent implementation of fast sockets and through a special user-level protocol.

In both cases, the implementation is done automatically by PeakWare based on the graphical description of the communication behavior of the application. At the end, the NEPHEW environment will generate an optimized application fully exploiting the capabilities of a SAN interconnected commodity cluster with minimal efforts on the side of the end user. This will open the cluster architecture to a whole range of new users and applications accelerating the acceptance of clusters also in the none computer science community.

The remainder of this paper is organized as follows: Section 2 introduces the NEPHEW project and its partners. Section 3 then describes the project's core, the PeakWare tool set. In Section 4, one of the three NEPHEW applications is introduced in detail, followed by the implementation strategy using PeakWare in Section 5, and first experiences on top of the Windows NT cluster in Section 6. The paper is then rounded up in Section 7 with a brief outlook on future work and some concluding remarks.

2 Background for NEPHEW

NEPHEW is a EU funded project in the 4th ESPRIT framework programme. It brings five partners from industry, research, and academia together that have the strong competence not only to successfully implement a comprehensive and efficient tool environment for SAN connected clusters, but also its thorough evaluation. The consortium, whose structure is depicted in Figure 1, consists of Dolphin ICS, the leading manufacturer of SCI technology, Matra Systems &

Information, the provider and developer of PeakWare, and three end users with high–performance cutting–edge applications, namely ELCO Sistemas, Joanneum Research, and the Technische Universität München.

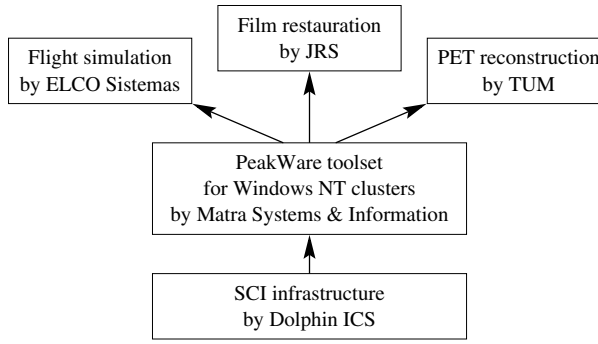


Fig. 1. The structure of the NEPHEW consortium.

The core of the NEPHEW project, the actual toolset is provided by Matra Systems & Information. It based on an earlier version for embedded real–time multiprocessor systems called PeakWare [10]. Within the NEPHEW project, Matra is porting this toolset to Windows NT and tailoring it to the specific requirements in general purpose cluster environments. This includes easing the user interface, removing specifics for real-time environments, and adding additional support for adding and removing cluster nodes. In addition, the back-end of PeakWare is adapted to the new hardware environment by adding support for Win-Sockets and high-performance communication via SCI.

The latter one is achieved in a close cooperation with Dolphin ICS. They bring their extensive low-level expertise about SCI hardware and low-level software into the consortium guaranteeing a highest possible performance of the final solution. As described above, two separate ways are envisioned to achieve this goal: through a high-performance socket implementation on top of Windows 2000 or through a specific low-level implementation taking direct advantage of SCI's hardware distributed shared memory (DSM). While the first approach provides a maximum of portability and reliability, the second approach delivers maximal performance. The right tradeoff between these two approaches can be done by the application programmer through easy graphical annotations within PeakWare to tailor the generated code to the application's specific needs and requirements.

For the evaluation of the PeakWare/NEPHEW toolset the consortium includes several end user application providers from different areas and backgrounds. ELCO Sistemas, a Spanish producers of flight simulation systems, will use PeakWare for the efficient implementation of low-cost PC based helicopter simulators. These systems are intended to close the gap between expensive spe-

cialized embedded simulators and game-like single systems programs and hence also allow small companies to gain access to serious and realistic simulation environments. Joanneum Research, an Austrian research institute, is specialized on the restauration of historic film material that has deteriorated after long storage. This is a very compute intensive application and requires efficient parallelization.

The third application, which is provided by the Technische Universität München, more specifically by a cooperation between the "Nuklearmedizinische Klinik und Polyklinik" [2] at the department of medicine and the "Lehrstuhl für Rechnertechnik und Rechnerorganisation" [3] at the department of Computer Science, comes from the area of Positron Emission Tomography and implements a necessary postprocessing step, the reconstruction of PET images from raw scanner data with a high image quality. This application will be used in this work as the example on how the NEPHEW toolset can be applied and is therefore described in more detail below in Sections 4 and 5.

3 PeakWare: Toolset for Efficient Cluster Computing

PeakWare [10] is a software product by Matra Systems & Information that was originally targeted to program real-time multiprocessor systems like the ones from Mercury Inc. [9]. In the NEPHEW project the system is ported to Windows NT and adapted to the specific needs in cluster environments.

Application development in PeakWare is generally done in five steps. First the data flow behavior has to be extracted from the overall application design and the main communication paths have to be determined. This analysis forms the basis for the decomposition of the application into modules, one of the central concepts of PeakWare. Modules are logical units of functionality that form the basis for global communication. The separation into these modules has to be done in a way that all communication between the modules can cleanly be specified.

The information gained through this analysis is then used to graphically describe the communication behavior in the so called software graph. An example with two modules and bidirectional communication is shown in Figure 2. PeakWare also offers the ability to scale individual modules, i.e. to replicate and distribute them among the cluster. This concept offers an easy way to introduce data parallelism into an application and allows the easy scaling on potentially arbitrary numbers of nodes.

Each module consists of several functions with the global communication channels as input and output arguments. The implementation of the functions themselves is done in external source files using conventional sequential programming in C. The source code can be augmented by macros defined by PeakWare in order to establish a connection between the functions and the PeakWare system.

The next step is the definition of the hardware that is supposed to be used for the application. This is again done with a graphical description, the hardware graph. An example of such a graph can be seen in Figure 3. It shows a small

[2] Clinic for nuclear medical imaging
[3] Chair for computer technology and organization

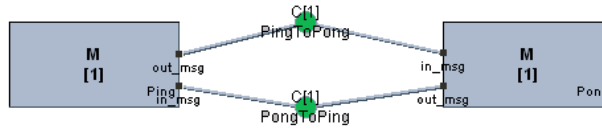


Fig. 2. Example of a PeakWare software graph (simple ping-pong)

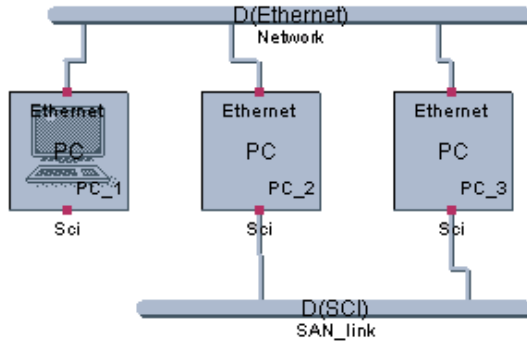


Fig. 3. Example of a PeakWare hardware graph (2 node cluster)

cluster of two compute and one host node connected by Fast Ethernet and SCI (between the two compute nodes). For any communication between two nodes connected by more than one network, PeakWare then selects the appropriate one based on the user's choice of protocols for the particular communication path. These parameters, however, are defined in the software graph leaving the specification of the hardware graph fully independent of the software graph. This enables the option to change the hardware in terms of node description and/or number of nodes without requiring changes in the software graph.

Once the hard- and software graph have been completed, PeakWare gives the user the option to specify a mapping between modules (from the software graph) and nodes (as specified in the hardware graph). This mapping defines which module is executed on which node and hence represents the connection between the two graphs. It also allows the easy retargeting of applications to new hardware configurations as well as a simple mechanisms for static load balancing.

The last step, after the mapping has been done and all routines have been implemented in external source files, is the code generation. In this process PeakWare uses the information from the graphical description of the software and hardware graphs and generates C source code that includes all communication and data distribution primitives. This code can then be compiled with conventional compilers resulting in a final executable and a shell script to start the application on all specified nodes; no further user intervention is necessary resulting a very easy-to-use environment.

4 Nuclear Medical Imaging Using PET

This easy-to-use programming environment is evaluated within the NEPHEW project using three large–scale real–world applications from a wide range of domains. One of them comes from the area of nuclear medical imaging and implements the reconstruction of Positron Emission Tomography (PET) images, a necessary post-processing step in the daily clinical routine when working with nuclear imaging.

Positron Emission Tomography is a nuclear medicine technique which allows to measure quantitative activity distributions in vivo. It is based on the tracer principle: A biological substance, for instance sugar or a receptor ligand, is labeled with a positron emitter and a small amount is injected intravenously. Thus, it is possible to measure functional parameters, such as glucose metabolism, blood flow or receptor density. During radioactive decay, a positron is emitted, which annihilates with an electron. This process results in two collinear high energy gamma rays. The simultaneous detection of these gamma rays defines lines-of-response along which the decay occurred. Typically, a positron tomograph consists of several (e.g. 32) detector rings covering an axial volume of 10 to 16 cm. The individual detectors are very small, since their size defines the spatial resolution. The raw data are the line integrals of the activity distribution along the lines-of- response. They are stored in matrices (sinograms) according to their angle and distance from the tomograph's center. Therefore, each detector plane corresponds to one sinogram. Image reconstruction algorithms are designed to retrieve the original activity distribution from the measured line integrals. From each sinogram, a transverse image is reconstructed, with the group of all images representing the data volume.

Ignoring the measurement of noise leads to the classical filtered backprojection (FBP) algorithm [4]. Reconstruction with FBP is done in two steps: Each projection is convolved with a shift invariant kernel to emphasize small structures but reduce frequencies above a certain limit. Typically, a Hamming filter is used for PET reconstruction. Then the filtered projection value is redistributed uniformly along the straight line. This approach has several disadvantages: Due to the filtering step it yields negative values, particular if the data is noisy, although intensity is known to be non-negative. Also the method causes streak artifacts and high frequency noise is accentuated during the filtering step.

Iterative methods were introduced to overcome the disadvantages of FBP. They are based on the discrete nature of data and try to improve image quality step by step after starting with an estimate. It is possible to incorporate physical phenomena such as scatter or attenuation directly into the models. It is generally acknowledged that iterative methods yield better images in low count situations.

On the down side, however, these iterative methods are very computational intensive. For a long time, this was the major drawback for clinical use of these methods, although they yield improved image quality. Speedup of iterative reconstruction is gained by the use of algorithms which use only subsets of data within one iteration (Ordered Subset Expectation Maximization, OSEM) [5]. Still, due to the iterative nature a substantial amount of time is required for the

reconstruction of one image plane. Thus, parallel processing of iterative reconstruction would help improving medical imaging.

5 PET Image Reconstruction Using NEPHEW

The usual way to implement PET image reconstruction on top of loosely coupled architectures like clusters is based on the master-slave-principle. With image reconstruction it is quite easy to distribute the computational load on the different machines, as an image consists of a fixed amount of planes (typically 47 or 63 depending on the scanner type). For each pixel of these planes the same arithmetic operations are executed. The basic approach is therefore to distribute the raw scanner data from the master plane by plane to the slaves running on the different nodes. Each slave process reconstructs one plane of the image and sends the result back. The master is then responsible for finalizing and storing the reconstructed image.

Using the modular design of PeakWare a slightly different approach has to be done: Three modules are used for reconstructing the image. A sender-module distributes the planes to a scaled consumer module, which corresponds to the slave in the approach described above. The planes are distributed in a round-robin fashion and one plane at a time. After a consumer instance has reconstructed a plane the receiver module is informed, which plane was reconstructed. In addition the sender is informed, that a next plane can be sent. The sender distributes the planes of the image until the reconstruction of all image planes has been acknowledged by the receiver. If no planes are left, planes might be resubmitted to idle consumers resulting an easy, yet efficient fault tolerance scheme with respect to the consumer modules.

The software graph based on the design described above, is depicted in Figure 4. It shows the three modules, with the consumer module scaled to $\$(SCALE)$ instances. In addition, the main data paths form the sender through the consumer to the receiver is visible in the middle augmented by the two acknowledgment paths leading back to the sender.

The software-graph with the three modules can be mapped on to arbitrary hardware graphs. This enables the easy distribution of consumer module instances across arbitrarily scaled clusters. It is also possible to change the number of nodes without changing the software-graph by simply remapping it to a different hardware-graph. This independence of hardware and software description drastically eases the port of application between different hardware configuration and allows for an easy-to-handle scalability for the PET reconstruction application.

6 Preliminary Experiences on Windows NT Clusters

The project is currently in an early stage of development. Therefore, only limited results can be presented here. They, however, give a good first impression of

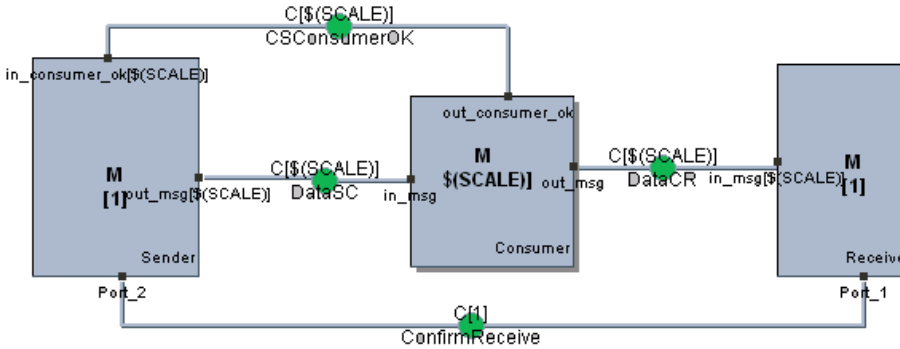


Fig. 4. Software graph for the PET reconstruction application.

the direction the project and allow a first insight into the capabilities of the NEPHEW approach.

In the first phase of the project, a prototype of the communication framework of the PET reconstruction application has been designed and implemented using PeakWare. Its software graph is shown above in Figure 4. The development of this prototype clearly showed the potential of the NEPHEW approach. The actual routines that had to be hand coded are extremely small only limited to the core functionality. Any required communication handling, which in standard message passing environments contributes to a majority of the code size, is created automatically by PeakWare, leaving the programmer with convenient and easily understandable code segments.

In addition, the orthogonal design of the hardware and software graphs, in addition with the extensive mapping capabilities of PeakWare, allows an easy application prototyping on smaller clusters or even single node machines. Once completed, the final application can easily moved to a target cluster of arbitrary size by remapping the modules. This significantly reduces the implementation time and lowers the usage threshold for non computer science users.

First concrete performance results have been achieved in the low-level communication layers provided by Dolphin. The performance data is summarized in Table 1 [1], which compares three different approaches. All numbers are gathered on top of Windows 2000 using a high–end PC cluster with a 64 bit PCI bus and the 64 bit PCI adapter D320 by Dolphin.

The first two options provide a socket-like interface to the user, one time through a standard network driver within the Windows kernel (NDIS) and one time using a fast socket library with user level communication transparently embedded into the overall system through a so called fast socket switch[4]. This convenient API for the user, however, comes at the price of reduced performance

[4] This switch allowing the transparently *switching* between standard Ethernet and fast communication networks is currently developed at Microsoft and will be included in the Windows 2000 Data Center.

caused by the overhead of the additional software layers. Especially latency is drastically increased. In addition, the NDIS packages pays an extra price for being a kernel component further reducing the performance. It is worth noting, however, that these numbers are only first experiments and better performance is expected for the final product.

In any case, however, when optimal performance and lowest possible latency is required, SCI has to be deployed using its native HW-DSM interface by performing communication directly on shared memory segments. In this case a bandwidth of up to 118 MB/s and a latency as low as 2 μs can be achieved.

	Bandwith	Latency
SCI over NDIS	62 MB/s	105 μs
SCI over Fast sockets	90 MB/s	75 μs
SCI raw	118 MB/s	2 μs

Table 1. First experimental results of SCI as the interconnection network.

In summary, SCI provides state-of-the-art performance to the user. Depending on the needs of the application the user has the choice of either relying on the existing convenient messaging standards, like Winsockets, or defining their own protocols and getting access to optimal performance. PeakWare will be able to use any of these communication mechanisms providing the user with the optimal choice for their particular application. Experiences with earlier versions of PeakWare on Mercury system have shown that the system due to its lean and efficient implementation only adds a very small latency penalty to the raw communication times, delivering close to optimal performance to the end users.

7 Conclusions and Future Work

With the availability of clusters to more users than ever before, it is necessary to provide easy-to-use, high-level programming environments. Only this will give also give unexperienced end users and non computer scientists access to low-cost parallel computing. Within the NEPHEW project, such an environment, targeted towards Windows NT based commodity clusters connected with SCI, a state-of-the-art high bandwidth, low latency SAN, is under development. This approach is being evaluated within the project using three computational intensive application. One of these, the reconstruction of PET images, is used in this work to demonstrate the applicability of developed concepts.

First experiments with the PET application have proven the ease-of-use of PeakWare. The application development phase is drastically shortened as all the overhead associated with communication management is automatically being taken care of by the toolset. In addition, the PeakWare software graph allows for an easily understandable presentation of the application's behavior. This helps

to significantly lower the learning curve for parallel programming and reduces potential errors.

Besides these ergonomic experiences, this work has also shown the performance potential of the underlying architecture. SCI networks deliver high-performance for direct process–to–process communication. By embedding SCI into the PeakWare toolset, the SCI performance potential is made available to the end user in the easiest possible way.

The next steps in the NEPHEW project include, besides further tests with cluster hard- and software, the completion of the three NEPHEW applications. In the end, all three application partners will have a running and robust parallel solution of their application on top of PeakWare. The PET reconstruction code will then be used in daily clinical routine as a standard instrument for the medical personal to the benefit of the patients.

In summary, the NEPHEW framework will allow users the easy implementation of parallel applications for PC based clusters and the efficient utilization of high-speed networks like SCI. It will form the basis for many application ports from a large variety of application domains and therefore open the cluster architecture to wide range of new users and applications. This will further increase the attractivity of the cluster platform and accelerate their acceptance in daily production scenarios.

References

[1] T. Amundsen. SCI performance under Windows 2000. personal communication with Dolphin ICS, May 2000.

[2] A. Bengelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. *A User' s Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083, July 1991.

[3] H. Hellwagner and A. Reinefeld, editors. *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *LNCS State-of-the-Art Survey*. Springer Verlag, October 1999. ISBN 3-540-66696-6.

[4] G.T. Herman. *Image Reconstruction from Projections*. Springer-Verlag, Berlin Heidelberg New York, 1979.

[5] H. Hudson and R. Larkin. Accelerated image reconstruction using ordered subsets of projection data. *IEEE Transactions on Medical Imaging*, 13:601–609, 1994.

[6] IEEE Computer Society. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.

[7] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org.

[8] WWW:. Welcome to Giganet . http://www.giganet.com/, May 1999.

[9] WWW:. Mercury Computer Systems (NASDAQ:MRCY) Home Page. http://www.mc.com/, January 2000.

[10] WWW:. Peakware — Matra Systems & Information. http://www.matra-msi.com/ang/savoir_infor_peakware_d.htm, January 2000.

SEEDS: Airport Management Database System

Tomáš Hrúz[1,2], Martin Bečka[3], and Antonello Pasquarelli[4]

[1] SolidNet, Ltd., Slovakia, www.sdxnet.com,
tomas@sdxnet.com
[2] Department of Adaptive Systems, UTIA,
Academy of Sciences of the Czech Republic, Prague[‡]
[3] Mathematical Institute, Slovak Academy of Sciences,
P.O.Box 56, 840 00 Bratislava, Slovakia,
becka@ifi.savba.sk
[4] Alenia Marconi Systems, Via Tiburtina km 12.4, I-00131 Rome,
apasquarelli@aleniasystems.finmeccanica.it

Abstract. The article describes an airport database management system, which is a part of large simulation environment developed under the ESPRIT project SEEDS. Airport management database is a distributed computing system written entirely in Java and connected to the rest of the SEEDS system through CORBA interface. The majority of the SEEDS modules is written in C++ and communicates through CORBA. The airport management database consists of an application server, three different clients and a CORBA connection module.

1 Introduction

This article describes an Airport Management Database (AMDB) system which is a part of larger simulation environment SEEDS [1,6].

SEEDS is a distributed simulation environment composed of powerful workstations connected in local network, suitable to evaluate advanced surface movement guidance and control systems, to validate new international standards and to train operators.

The aim of AMDB software module is to describe various external aspects of the core airport simulation model like meteorological situation and its changes, flight data list (FDL) and its changes, Initial Climbing Procedures (ICP), Instrument Approach Charts (IAC), Standard Instrumental Departure (SID) and Standart Approach Route (STAR) description and visual information. To emphasize the external aspect of AMDB with respect to the core simulator we sometimes use the term external world model instead of AMDB.

The AMDB module is designed to contain a relational database subsystem to handle large data sets and a three level architecture (SQL server [7], application server and clients) to achieve high level of flexibility. Another aspect of AMDB module design is a wide area network operation emphasis, which is leading to the architecture centered around the Java system.

[‡] Grant GACR 102/1999/1564

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 861–868, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The core of the SEEDS simulator is written in C++ and uses the CORBA standard to communicate between different system modules. The AMDB module is entirely written in Java. This situation provides an excellent occasion to test the cooperation between Java and CORBA based C++ subsystems.

In particular, the application server (APS) is written in Java to see whether the speed of recent Java virtual machine (JVM) implementations is able to cope with the tasks arising in such complex applications.

As far as SEEDS together with AMDB module contains various hardware and software platforms we have decided to use only open software tools for which we had the source code. This approach has two reasons:

1. In very complex and multi-platform systems we consider the absolute control over the development process provided by source code as inevitable for any high quality software.

2. Open software systems usually contain a less powerful visual development tools and we were interested in knowing whether these high level visual features are really necessary for development of very complex systems.

The article is organized as follows: in Sec. 2 we describe architecture, design specification and basic features of the AMDB system. Then we conclude with some notes and experiences drawn from this work.

2 Airport Management Database System

The information stored in the airport management database can be structured to groups according to the importance and complexity. In the SEEDS airport model architecture [1] the following navigation data have been considered for inclusion to the external world model. (1) Meteo information, (2) FDL and (3) SID, STAR, ICP and IAC charts. These information groups are implemented in the AMDB module through relational database model, APS and clients. The relational database stores the information about the navigation data in the structured way and APS is responsible for all non-relational model aspects. There are three main clients in the system.

1. Meteo client responsible for meteo information rendering and changes.
2. Flight data list client responsible for the FDL processing.
3. Image viewer client responsible for SID, STAR, ICP, IAC charts previewing and processing.

The synchronization, authentication and all other aspects of AMDB allow arbitrary number of clients (limited only by system resources) of each type running at the same time.

2.1 Architecture

Today's computing environment enforces definitively a server/client architecture. The main reasons are flexibility, scalability and reusability. Moreover, the AMDB module architecture is Java (network computing) oriented and has a

SQL relational database in the background to meet the design goals. The client Java classes communicate with the SQL database engine through JDBC layer and driver [8].

However, an architecture conforming to the above mentioned points can be designed in various ways. Let us consider first the architecture in Fig. 1 but without the third quadrant containing AS. With respect to the relation between SQL database and clients this can be called two level architecture because there are two levels: 1.clients and 2.SQL servers. The advantage of such architecture is simplicity, a disadvantage is limited modeling strength and flexibility.

With modeling strength and flexibility we mean the following concept. Any software system can be considered as a model of some process. If we consider a two level architecture as above, we can efficiently construct a model for a process which can be well represented in terms of relational database. However, if the modeled process does not fit well in this class we have to add software modules which represent its non-relational structures. If a two level architecture is used these modules must be split between the SQL server (as embedded procedures) and the clients. Such splitting can be very inefficient from the software design point of view as well as from the final system efficiency point of view. As a solution to the above problem a concept of three level architecture can be used. The three level architecture has: 1. clients, 2. application servers, 3. SQL servers.

Following the ideas above, all non-relational aspects of the model are located in the APS. It is also responsible for managing and forwarding the database queries of clients to the SQL server. In the SEEDS project an absolute necessity of APS comes from the fact that any change of the database state must be transmitted to the rest of the system in form of events, so that all modules can change their behavior according to the new state of the external world. The final three level architecture used for AMDB is shown in Fig. 1 where APS is designed in Java and is called from the clients through Java Remote Method Invocation (RMI). Another advantage of this architecture is that design of such system contains also a task of application protocol development. Moreover, in the case of very heavy server computation related to some non-relational modeling aspects, the server class can call C or C++ libraries through JNI (Java Native Interface).

The airport management database architecture and its relation to the processing of events in SEEDS is illustrated in Fig. 2 on an example of the meteo data processing subsystem. AMDB calls a meteo event server written in C/C++ with a CORBA interface, which is a part of core SEEDS system. The SEEDS modules register to the meteo event server for a particular class of events. The major events with respect to the external world model are changes in the model database state.

For example an operator wants to change the visibility at some airport object. He starts a master mode of client module, which allows him to change the database state. The request is forwarded through RMI to AS, which generates a SQL operation to the SQL server and a notification event to the meteo event

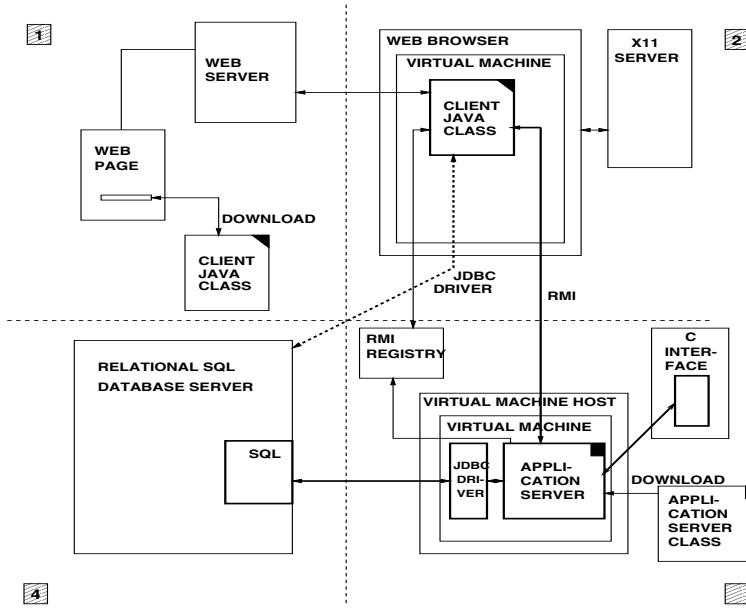


Fig. 1. The Java client/server 3 level architecture for Airport Management Database

server. APS notifies also all registered external world clients about the change so that all other operators see the change on the external world client screen.

Because the event server has been notified all SEEDS modules registered for this event class will obtain a notification about the event. Then they can ask through a CORBA interface of the application server about the new values of AMDB and they can change the behavior according the new data. For example the scenario generation module can change the picture on appropriate operator and pilot screens etc.

To summarize the architecture, the AMDB system consists of the following subsystems: **(a) SQL server.** The SQL server used is PostgreSQL [7], which is running on most UNIX platforms. **(b) SQL relational database.** The AMDB use a database maintained by SQL server. The relational database consists in a set of tables, which provide a relational model of AMDB subsystems like airports, meteo data, navigation procedures etc. **(c) Web server and web pages.** The client applets are stored and transmitted through the Apache web server. **(d) The clients.** The code of clients is stored in signed archives on the web server. Clients are downloaded from the web server to the browsers where they are run on browser virtual machine. **(e) The application server.** APS is running as a Java application under JVM.

In the prototype configuration the application server, the web server and the SQL server are running on Alpha station with Digital 64bit UNIX.

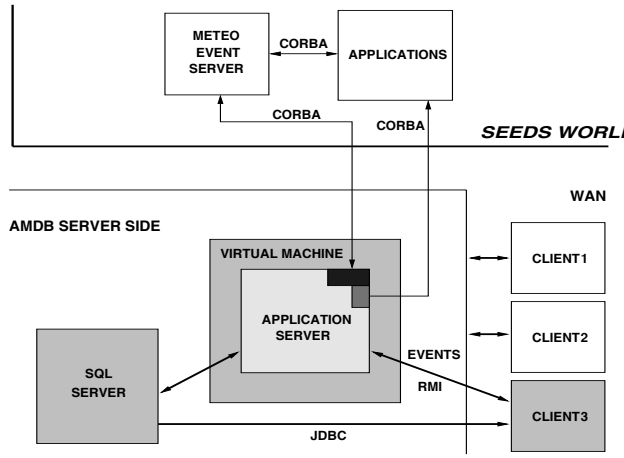


Fig. 2. External World Model Architecture and Communication. The communication pattern between application server, SQL server and clients in the AMDB module is shown as the emphasized triangle structure

2.2 Data Transmission Rules

The SEEDS system is highly heterogeneous and contains more platforms (e.g. Microsoft NT on Intel, UNIX on Silicon Graphics, UNIX on Alpha, Java etc.) therefore it is necessary to establish communication rules which allow robust and efficient communication between the modules and platforms. For the AMDB module design we have identified the following rules:

(1) All aspects of the system, which can be modeled by relational database are modeled by the database structure and are centered in SQL server.

(2) If a client is interested in data which are stored in the database and do not require non-relational processing they read the data directly from the SQL server as is illustrated in Fig. 2 with an emphasized triangle pattern.

(3) The only agent allowed to write the data to the SQL database is the application server.

(4) All non-relational modeling and processing is centered in application server. In the case of the AMDB module this means for example notification and data transmission between the AMDB module and the rest of SEEDS.

(5) Because of the heterogeneous character of SEEDS it is preferable to convert most of the data which will be transmitted over the network to integer formats. This is an obligatory rule for primary key data. It means that all information entities are coded as integers and only these codes are sent over the network. The clients are responsible for transformation between the integer representation and other forms of representation like strings etc. For this purpose the clients call directly the SQL server.

2.3 Communication with SQL Server

Following our open software strategy we have used the PostgreSQL [7] database system. The resulting experience is very positive. The system is robust, stable and efficient. Moreover, it defines a very rich set of database types. On the other hand, the main disadvantage of this system, when compared with commercial systems like Oracle SQL servers is a certain lack of preprocessing, postprocessing and visual tools. However, the minimal price (only the maintenance cost) and the open character of this system have far outweighted its disadvantages especially in a research and development project as is SEEDS.

The clients and APS communicate with the SQL server using JDBC layer defined by Java system [8] which in turn calls JDBC driver which is provided by SQL server manufacturer, in our case by PostgreSQL system.

2.4 Security Model

The AMDB module is designed as a network computing structure therefore the security aspects are very important.

The PostgreSQL server is well equipped with security options ranging from a simple password authentication to KERBEROS and data channels encryption. To choose the appropriate option is a matter of configuration depending on the actual AMDB usage.

APS is written in Java as a Java application. One of the specification and design problems here is related to the fact that Java version 1.1, which is still not widely accepted by the browser industry does not contain interface to CORBA. This is provided in a new generation of Java, version 1.2. Therefore, we have adopted the following security design rules illustrated in Fig. 3.

- The application server is written as a Java application.
- The server code is conformant to the Java 1.1 standard.
- The server is compiled and running in the Java 1.2 system. The security restrictions are defined with a special security manager written for the AMDB module.
- The clients are written as Java applets.
- The code of clients is conformant to the Java 1.1 standard.
- The clients are compiled and running in the Java 1.1 system. The security restrictions are handled according to the Java 1.1 system. This means that we generate signed archives stored on a web server.

2.5 Application Server and Clients

APS is responsible for all non-relational modeling aspects. After successful reconstruction of the initial state an object for the SQL communication is constructed, which opens a channel to SQL database server used for data processing. This SQL channel is used to read and write the data from the database during the

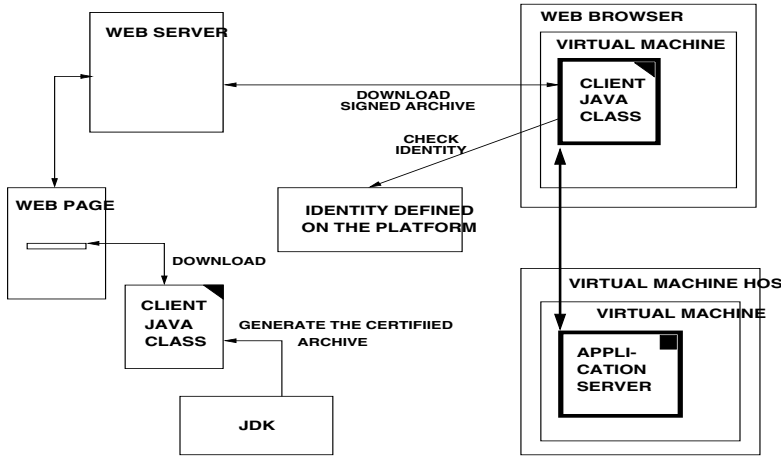


Fig. 3. Illustration of the client security model

whole life period of APS. All other agents are allowed (even sometimes obliged) to read directly from SQL server but not to write.

Clients connect to APS through well defined "channels" which are defined as two sets of remote procedures. One set for each direction. RMI connection of a client is initialized through a well known remote object. The reference to the remote object is obtained through the RMI naming services. Then a private channel is constructed which contains private remote references to the procedures above. APS maintains a list of channels which is periodically checked for dead clients. The channels for dead clients are deleted so that there is no resource deficit during a long server runs. The channel maintenance subsystem use a watch dog mechanism where in regular intervals each client must call the following time synchronization procedure:

Timestamp watchDog(Timestamp clientSimulationTime)

through which the client sends its value of the simulation time and obtains back the server value of the simulation time. If they differ in larger value, the client is obliged to synchronize with APS. At the same time, when APS obtains such call it updates the watch dog structure. This mechanism provides a means how to detect dead channels as well as synchronize simulation time between the clients and APS.

We have implemented three main AMDB clients, which are connected to the database through JDBC. During the client run it is necessary to load information from the database concerning the user identity tests, simulations, airports, aircraft types, airport objects and its sites, predefined meteo values, airport map images, etc. Some of these data are loaded once and cashed in the application, other ones are dynamically loaded when they are needed.

We have introduced a concept of two edit modes of the client, a slave mode and a master mode. In the slave mode a user can only view all information the client offers. In the master mode the user can change the data. The slave mode is

default, for switching to the master mode the user's identification and password is required. User is notified about the number of masters working on the same configuration, however users can concurently write to the database.

3 Conclusions

Concerning the software engineering point of view there are three main conclusions (experiences) from the SEEDS project. One concerns CORBA and the other two Java and open software systems.

CORBA together with Internet proved to be extremely efficient in development and integration of large systems which contain different platforms and geographically distant development teams. The introduction of new platform (Java) into the project has stressed the necessity to stick very closely to CORBA standards, otherwise the reusability of the software modules can be decreased.

We used Java extensively in AMDB design to test its reliability and efficiency. Concerning its complexity, Java is surprisingly matured even for industrial applications. We where running the application server on different platforms (Sun, Alpha) with very good results concerning speed and reliability.

The results with open software tools are very positive and the possibility to inspect the source code in some situations has far outweighed the disadvantage of not having a strong visual development tools. This do not indicate that they are not needed in current software development industry but it might indicate that their role is little bit overestimated and that the complexity level on which they are really inevitable lies higher as is usually considered.

In some situations it was necessary to change or correct the source code of systems which we used and this was far more efficient solution as to circumvent the problems with other means.

References

1. Bottalico, S., de Stefani, F., Ludwig, T., Rackl, G. : SEEDS – Simulation Environment for the Evaluation of Distributed Traffic Control Systems. In: Lengauer, Ch., Griebl, M., Gorlatch (eds.): Euro-Par '97, Parallel Processing, LNCS 1300, Springer-Verlag, Berlin Heidelberg New York (1997), 1357–1362
2. Bottalico, S. : SEEDS (Simulation Environment for the Evaluation of Distributed Traffic Control Systems). A Simulation Prototype of A-SMGCS. In: Proceedings of the International Symposium on Advanced Surface Movement Guidance and Control System, Stuttgart, Germany, 21-24 June 1999
3. Hrúz, T., Bečka, M. : Airport Management Database. SEEDS Internal Documentation SEEDS/MISAS-WP4-D4.6A-REV1-SW, 1999
4. Lewis, G., Barber, S., Siegel,E. : Programming with Java IDL. John Wiley & Sons, 1998
5. Flanagan, D. : Java in a Nutshell. O'Reilly, 1997
6. SEEDS official home page, http://www.lti.alenia.it/EP/seeds.html
7. http://www.postgresql.org, 1998–1999
8. http://www.javasoft.com, 1998–1999

HIPERTRANS: High Performance Transport Network Modelling and Simulation

Stephen E. Ijaha, Stephen C. Winter, and Nasser Kalantery

Centre for Parallel Computing, Cavendish School of Computer Science,
University of Westminster, 115 New Cavendish Street, London W1M 8JS, UK
{ijahas, wintersc, kalantn}@wmin.ac.uk
http://www.wmin.ac.uk

Abstract. HIPERTRANS (High Performance Transport network modelling and Simulation) is a fast and visually representative simulator that can predict traffic on a given urban road network. It was designed using object-oriented techniques and by re-engineering established road traffic models. It has the capability of interfacing to a range of UTC (urban traffic control) systems. Its high performance version was implemented by using a novel parallel programming platform called SPIDER; it has the capability of executing faster than real time and runs on distributed processors. The simulator provides a powerful GUI (graphical user interface) for entering road network models, configuring simulation runs, and visualising simulation results. The system provides helpful traffic diagnostic tools enabling local transport authorities, policy makers, researchers, and UTC manufacturers to gainfully exploit its functionality. The HIPERTRANS project was funded by the European Commission under the 4th Framework programme.

1 Introduction

1.1 The HIPERTRANS Requirements and Specifications

Traffic congestion problems in urban areas have prompted governments, local authorities and UTC manufacturers to be interested in a modelling and simulation tool that can be used to predict traffic situations. Such a tool is needed to achieve real-time traffic control measures and assist policy makers to make informed decisions [1]. The HIPERTRANS project conceived the idea of a fast, representative, flexible and visually comprehensive traffic simulation tool that can be achieved through the application of low cost but high performance computing environments, advanced simulation technologies, and industrial software production methodologies [2]-[3].

The HIPERTRANS simulation system consists of a new range of facilities for transport consultants, researchers, traffic engineers and UTC centre managers. The tool can enable transportation network operators to assess the performance of their road network quickly under a variety of operational conditions and behavioural patterns. Additionally, it can be used as a tool for hardware commissioning and operator training. Using event driven approach, the HIPERTRANS system is capable of putting vehicles into a network and representing the movement of individual vehicles.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 869-874, 2000.
© Springer-Verlag Berlin Heidelberg 2000

This paper describes the work carried out within HIPERTRANS during April 1997 to June 1999. After the description of the project in this Section, the next Section states the objectives of the work. Section 3 describes the methodology and approach employed by which these objectives were achieved. Section 4 provides the results achieve. The conclusions and recommendations for future research are presented in Section 5.

1.2 HIPERTRANS Partnership and Test Sites

HIPERTRANS, an EC-funded consortium, was led by the University of Westminster (UoW) and had 8 partners spread across 4 different EC countries. The partners participated within two broad groups: Technology Group and User Group. The 3 Technology Group partners came from 2 universities □Facult□s Universitaires Notre-Dame de la Paix □Namur (FUNDP) in Belgium and UoW in the UK; and a research institution - Institut National de Recherche en Informatique et Automatique (INRIA), in France. They were responsible for the definition, design, building, testing, and verification of the simulators that were developed within the project. The 5 User Group partners were formed by: 2 transport consultants - BKD Consultants Ltd (BKD), and W.S. Atkins (WSA), both in the UK, 2 UTC (urban traffic control) system manufacturers - Electronic Trafic S.A. (ETRA), in Spain and Peek Traffic Limited (PTL), UK, and a software house - SIMULOG in France. The User□s main role was to validate and demonstrate the simulators in a real traffic environment using real traffic scenarios.

The work on the real-time simulator and predictor made enormous use of two different kinds of proprietary UTC systems kindly made available to the project by two of our partners: SCOOT [4] partly owned by PTL; and STU (Sistema de Trafico Urbano), [5] wholly owned by ETRA. SCOOT is the predominant type of UTC system in the UK and STU in Spain. The GUI developed for visualising the results and inputting road network data models into the simulator was developed by using Simulog□s OGL graphical library. This library is based on the ILOG [6] system and is available on Windows and Unix platforms.

Two test sites, Hyde in the UK and Valencia in Spain were employed by the project. The criteria used in selecting these test sites were that they had the UTC systems used by our User partners and whose local authorities were willing to participate in the evaluation of the project. The participation of different UTC manufactures and local authorities in this project is also helping to overcome the challenges of non-interoperability of UTC systems not only in Europe but also throughout the world.

2 Objectives

The key aim of the project was to produce microscopic modelling and simulation tool for the easy and cost-effective development of road traffic control systems and for the effective management of traffic flow on road networks. Specific aims were to:

- develop a simulator able to interact with UTC systems at real-time speeds and develop a predictor consistent with users□requirement for look-ahead;
- implement a low-cost high-performance system on a scaleable workstation cluster;

- demonstrate that the use of high-performance simulation can speed up and dramatically improve the operation of urban and inter urban transportation networks;
- compare the execution times performance of the simulation algorithms;
- model road traffic on a small-to-medium sized network; and study technology and best practice for extending the simulator to handle inter-urban situations.

In general, the HIPERTRANS system aimed to have the following basic functionality:

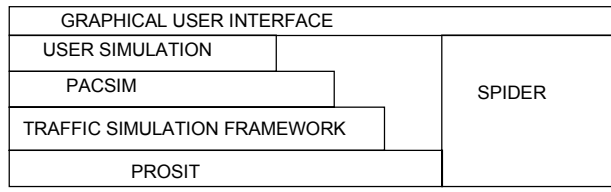
- performing microscopic simulation having a user-friendly graphical user interface;
- interfacing capability to UTC systems and capability of working in real time;
- being able to simulate and predict the state of the road traffic; and
- achieving high-performance computing on low cost platforms.

3 Technical Description

The implementation of the system was performed in three different but closely related stages involving the component software modules being developed on several sites and eventually integrated on a single platform. This provided essential co-ordination points within the software development and enabled the User Group to evaluate progress. They also assisted all partners to monitor and steer the development activities in an optimal manner. The strategy used in the work was geared towards applying results from the science base of advanced traffic modelling and control techniques as well as the technology of high-performance computing to achieve the simulators. The overall architecture consists of the simulator; a UTC system; and an interface between the simulator and the UTC system. The simulator is software and hardware configurable to:

- enable it to be used in real-time with different UTC types;
- provide a stand-alone simulation capability when not interfaced to a UTC;
- allow information relevant to a wide range of road networks to be used; and
- scale from low cost and moderate performance to higher cost higher performance.

The architecture shown in the Figure below is a combination of an object-oriented framework for traffic simulation, user simulation models, and application programming interfaces for interconnection to UTC systems. The background software used existed at three partner sites: PACSIM [7] developed by FUNDP; PROSIT [8] co-developed by INRIA and Simulog, and SPIDER [9] developed by UoW.



The HIPERTRANS Architecture

4 Results

The project has delivered a micro-simulator as specified, and was used for high-profile public demonstrations and evaluation at Hyde in the UK and Valencia in Spain [10]. The results provide a proof of concept of high-performance simulation as ascertained from the needs of users. The resulting simulator can run on a network of workstations, ensuring that simulation of large traffic networks can be achieved within a pre-determined real time limit. Starting with the current road conditions supplied by an operator, the simulator was shown to run much faster than real time.

A testing exercise [10] was carried out to check the simulator's correctness, and to evaluate its performance by comparing the outputs from the sequential version with those from the parallel version. These measured the speed-up when different numbers of processors were employed compared to when only one processor was used. The correctness tests were designed to prove that the algorithms employed by both versions do not alter the nature of the results when the simulation is run under identical conditions. The strategies used in the 'correctness' test involved making the size of the road traffic model a variable parameter while keeping the number of vehicles generated by each of the two versions constant. In the performance test, the number of vehicles generated by each of the two versions was varied while keeping the size of the road traffic model constant. The results of the tests for one processor and using 4 processors show the different execution times for the two versions both using a grid road network and a total simulation time of 35 minutes for different number of vehicles. When the number of vehicles becomes very large, the speed up may approach a limit that is approximately near the theoretical limit n where n is the number of workstations used in the distributed version. For realistic as well as meaningful evaluation, the correctness tests were carried out on a real road network in Nice, south France.

5 Summary and Conclusions

Through the application of low cost but high performance computing environments, advanced simulation technologies and industrial software production methodologies within the project, a fast, representative, flexible and visually interactive traffic simulation tool have become available. This paper has described the work and results of the HIPERTRANS traffic simulation system. The software architecture employed comprises: the object-oriented framework for traffic simulation, PROSIT, PACSIM, and SPIDER. SIPIDER enabled the capability of distributed simulation so that the software resides on a cluster of workstations and provides a sequentially consistent parallel execution. The speed of the simulator permits faster than real-time prediction of road network behaviour to be made. The simulator can enable UTC operators to examine the future effect of their selected actions on levels of congestion fast enough to be able to revise and re-test the performance before selecting the best action to take.

The HIPERTRANS project has proven that microscopic simulation can be one of the most effective tools in the design and management of road traffic systems. In comparison to related work, other useful simulation tools [11]-[14] have been developed in the past, but further advancement in high performance simulation

technology was motivated by the fact that large scale microscopic simulation are time consuming and so further research was carried out to look into the effective exploitation of distributed computing to speed up the simulation further in realistic road networks. This project has assured that the use of scalable parallel computing ensures that irrespective of the size and complexity of the modelled network, faster than real time criterion can be met.

Notwithstanding the advancement made within the HIPERTRANS project, more practical aspects need to be addressed. For future work, we recommend the development of an open simulation farmework which has the capability of integrating distributed executions with more user-friendly 3-dimensional graphical and animation interfaces, a wider variety of real-time UTC systems, and capability of receiving data from and sending data to traffic data and information systems. Future systems should not only be able to receive real-time traffic data and predict the future state of traffic, but should be able to incorporate emission models and GIS data. Other EC, national, and other research projects who have already built other types of simulators such as macroscopic simulators, emission models, and automatic debiting simulators, freight transport and crowd management simulators could benefit from HIPERTRANS results. It is possible for such other simulators to integrate with a microscopic road traffic simulator. Our collaboration with such builders will yield the possibilities of interfacing their work with our system. The purpose of such a modular and open simulator would be to use it in the area of transport, traffic and telematics where several opportunities are foreseen in typical applications such as traffic management, floating car data, and route guidance.

6 Acknowledgement

The HIPERTRANS project was funded by the European Commission's DG 7 under the Framework 4 programme. We are grateful to the EC and all the project partners.

7 References

1. HIPERTRANS Consortium, 1997, Deliverable D1: System Specification, *HIPERTRANS RO-97-SC-1005*, July 1997
2. IJAHA S.E. WINTER, S.C, KALENTERY N. and DANIELS B.K.: "HIPERTRANS: a road traffic simulation as an operational tool", Paper No. 46 in the 1998 *IEE International Conference on Simulations, University of York, UK,* 30 Sept - 2 Oct 1998.
3. SIEGEL, G, FURMENTO, N, and MUSSI, P.: "A Traffic Simulator for Advance Transport Telematics (ATT) Strategies" *IEEE conference Electrotechnological Services for Africa AFRICON'99,* 29/09/99 - 01/10/99, Cape Town, South Africa.
4. PEEK Traffic Systems, "SCOOT System Overview", 1995
5. ETRA I+D, 1996, STU-I+D-05-01, Version 3.1/1.2, Sistema de Trafico Urbano, Descripcion General STU.
6. HIPERTRANS Consortium: HIPERTRANS GUI and User's Manual, Annex 2 of Deliverable D5: Demonstration, *HIPERTRANS RO-SC-1005*, June 1999.
7. CORNELIS, E., and TOINT, P. L.: "An introduction to PACSIM: A new dynamic behavioural model for traffic assignment", *Report* 1997/98.
8. GAUJAL, B., and MUSSI, P.: PROSIT Manual and HIPERTANS: Object Oriented Simulation for Urban Traffic", *INRIA Internal Report,* 1995.

9. KALENTERY, N., SPIDER, *Internal Report, Centre for Parallel Processing, University of Westminster,* 1995
10. HIPERTRANS Consortium: Deliverable D5: HIPERTRANS Demonstration, *HIPERTRANS RO-97-SC-1005*, July 1999.
11. TSS: Transport Simulation Systems. *AIMSUN*. http://www.tss-bcn.com/aimsun.html
12. Rickert M. and P. Wagner. Parallel Real-time Implementation of Large-scale, Route-plan-driven Traffic Simulation. http://www.zpr.uni-koeln.de/GroupBachem/VERKEHR.PG/
13. PARAMICS: Traffic Simulation. *PARAMICS, see also* http://www-usa.quadstone.com/paramics/ext/index.html
14. Nuttall I, and Fellendorf M.: VISSIM for Traffic Signal Optimisation. Traffic Technology International '96. *Annual Review Issue*. 1996. pp190-2.

Topic 13
Routing and Communication in Interconnection Networks

Jose Duato

Vice-Chair

The papers in this workshop cover a wide spectrum, dealing with both theoretical and practical aspects. First of all, four regular papers propose improved strategies and models for collective communication, routing in irregular topologies, reliability estimation, and deadlock avoidance. These papers are summarized below:

- A Bandwidth Latency Tradeoff for Broadcast and Reduction, by Peter Sanders and Jop Sibeyn: This paper proposes the fractional tree algorithm, a new hybrid scheme for broadcasting and reduction, whose communication pattern interpolates between the sequential pipeline and pipelined binary tree algorithms. The paper also plots the reduction in broadcast latency over the sequential pipeline and pipelined binary tree algorithms in the absence of contention. The proposed broadcasting and reduction scheme is very interesting and can be directly included in communication libraries such as MPI. It significantly improves latency for relatively long messages.
- Improving the Up/Down Routing Scheme for Networks of Workstations, by Jose Carlos Sancho and Antonio Robles: This paper proposes new heuristic rules to compute the routing tables in NOWs with irregular topology. The paper also proposes a traffic balancing algorithm to obtain more efficient routing tables when source routing is used. Evaluation results show that the routing algorithm based on the new methodology increases throughput by a factor of up to 2.8 in large networks, also reducing latency significantly.
- A New Reliability Model for Interconnection Networks, by Rosa Alcover and Vicente Chirivella: This paper proposes a new reliability model for interconnection networks that considers all the fault patterns and their probability of occurrence in a tractable manner. The proposed model considers the behavior of the routing algorithm. This model uses Markov chains to model degraded network states. Fault patterns are grouped by considering the symmetry of the network and the fault patterns supported by the routing algorithm. Moreover, in order to make the analysis tractable, the number of degraded states is limited to those that have a non-negligible probability of being reached. The paper compares the behavior of the proposed reliability model with the simpler approach consisting of considering the worst case, showing the drastic increase in accuracy achieved by the proposed model.
- Deadlock Avoidance for Wormhole Based Switches, by Ingebjørg Theiss and Olav Lysne: This paper applies well-known deadlock avoidance techniques

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 875–876, 2000.
© Springer-Verlag Berlin Heidelberg 2000

to a new and interesting problem of building scalable "compound" switches from smaller "networked" switch modules in a way in which deadlocks are avoided. In doing so, the authors use extra logic in each switch module to perform flow control such that internal blocking within the switch is eliminated between different packet and control streams.

Six short papers complete the contents of this workshop. They mostly cover the theoretical aspects of the workshop, including analytical models and theoretical bounds. But some of the papers also cover performance evaluation and techniques to improve performance. These papers are summarized below:

- Probability-Based Fault-Tolerant Routing in Hypercubes, by J. Al-Sadi, K. Day and M. Ould-Khaoua: This paper describes a new fault-tolerant routing algorithm for hypercubes based on probability vectors. The proposed routing algorithm significantly improves over previously proposed schemes based on safety vectors.
- Performance Analysis of Pipelined Circuit Switching, by Geyong Min and Mohamed Ould-Khaoua: This paper proposes an analytical model of pipelined circuit switching for hypercubes. It supports virtual channels. The evaluation results show that it is accurate for low and medium network loads.
- Optimal broadcasting in even tori with dynamic faults, by Stefan Dobrev and Imrich Vrt'o: This paper obtains an upper bound for broadcast in faulty k-ary n-cubes with even k using the all-port model. The paper assumes that faults are dynamic, at most 2n-1 links are faulty, and faults are distributed in the worst possible manner.
- Experimental Evaluation of Hot-Potato Routing Algorithms on 2-Dimensional Processor Arrays, by Constantinos Bartzis, Ioannis Caragiannis, Christos Kaklamanis and Ioannis Vergados: This paper presents an application of a very well known routing scheme under static and dynamic packet generation. The authors state that the approach they are proposing would be very suitable for all-optical networks.
- Broadcasting in all-port wormhole 3-D meshes of trees, by Petr Salinger and Pavel Tvrdik: This paper proposes a broadcast algorithm for three-dimensional meshes of trees, assuming wormhole switching and the all-port model. The paper also proves the optimality of the proposed algorithm with respect to the number of steps or rounds.
- A Clustering Approach for Improving Network Performance in Heterogeneous Systems, by V. Arnau, J. M. Orduña, S. Moreno, R. Valero and A. Ruiz: This paper proposes a clustering technique to split networks of workstations with irregular topology into clusters. It also proposes a metric to assess the quality of process-to-processor mappings and experimentally shows the high correlation existing between the proposed metric and network performance.

Experimental Evaluation of Hot–Potato Routing Algorithms on 2–Dimensional Processor Arrays*

Constantinos Bartzis[1**], Ioannis Caragiannis[2], Christos Kaklamanis[2], and Ioannis Vergados[2]

[1] Department of Computer Science and Engineering
University of California, Santa Barbara, USA
[2] Computer Technology Institute and
Dept. of Computer Engineering and Informatics,
University of Patras, 26500 Rio, Greece.

Abstract. In this paper we consider the problem of routing packets in two–dimensional torus–connected processor arrays. We consider four algorithms which are either greedy in the sense that packets try to move towards their destination by adaptively using a shortest path, or have the property that the path traversed by any packet approximates the path traversed by the greedy routing algorithm in the store–and–forward model. In our experiments, we consider the static case of the routing problem where we study permutation and random destination input instances as well as the dynamic case of the problem under the stochastic model for the continuous generation of packets.

1 Introduction

We consider a form of packet routing known as *hot–potato* routing. The network is modeled as a directed graph where the nodes are the processors and the unidirectional edges are communication links between processors. Each processor has an injection buffer and a delivery buffer. When a new packet is generated, it is stored in the injection buffer of its source processor; when a packet reaches its destination processor, it is stored in the delivery buffer. The routing is performed in discrete, synchronous time steps. During each step, a processor receives zero or one packet along each incoming edge and must send all the packets it received out along outgoing edges with at most one packet leaving per outgoing edge. No buffers are required to hold the packets between the time steps. Any packet that arrives at a node other than its destination must immediately be forwarded to another node. The topology we consider in this paper is that of the 2–dimensional torus–connected processor array.

* This work was partially funded by the European Union under IST FET Project ALCOM–FT and RTN Project ARACNE. An extended version of the paper can be found at http://students.ceid.upatras.gr/~caragian/bckv00.ps

** Part of this work was performed while the author was at the Department of Computer Engineering and Informatics, University of Patras, Greece.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 877–881, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In static (or batch) routing problems, all processors generate a single packet simoultaneously. The running time of a routing problem is the number of time steps required to deliver all packets to their destinations.

In dynamic routing problems, each node continously generates packets with an injection rate λ. New packets are stored in the injection buffer and wait to be served. When a processor receives less than four packets along its incoming edges, it considers a packet from its injection buffer. When a packet starts moving, it is never buffered at any node until it reaches its destination, where it is stored in the delivery buffer and absorbed.

The first hot-potato algorithm was proposed by Baran [1]. Borodin and Hopcroft [4], Prager [13] and Hajek [8] presented algorithms for hypercubes. Hot–potato routing algorithms for 2–dimensional meshes and tori were proposed by Bar–Noy et al. [2], Ben–Aroya et al. [3], Newman and Schuster [12], Kaufman et al. [10], Feige and Raghavan [7], and Kaklamanis et al. [9]. All of them deal with batch routing problems. The only study of the dynamic case we are aware of is that of Broder and Upfal [5].

An important class of hot–potato routing algorithms is that of greedy algorithms. In these algorithms, each node forwards each packet closer to its destination whenever possible. Although greedy algorithms have been observed to work well in practice (for static routing problems), the known theoretical results for their performance are far from being tight (see Busch et al. [6]).

Especially for meshes and tori, a class of hot–potato routing algorithms that has received much attention is that of algorithms that make packets follow paths that approximate their natural greedy path (i.e., the path utilized by the greedy routing algorithm in the store–and–forward model [11]). Such algorithms were proposed and analyzed in [9].

2 Short Description of the Algorithms

In this section, we briefly describe four algorithms on the 2–dimensional torus network, namely the greedy algorithm, algorithm A1, algorithm KKR, and algorithm A2. The greedy algorithm is a variation of the folklore algorithm mentioned in the bibliography. Algorithm KKR was proposed in [9]. Algorithm A1 is a simple algorithm that "approximates the greedy path" while algorithm A2 is a variation of algorithm KKR. To our knowledge, algorithms A1 and A2 have not been studied in previous work.

The greedy algorithm. The greedy algorithm which was implemented tries to move packets towards their destination by adaptively using shortest paths, also trying to minimize the difference between the horizontal and the vertical distance of each packet from its destination.

The decisions the algorithm makes are local, since they depend on the destination of the incoming packets and the order in which these packets are examined. The implementation of the algorithm obeys the one–pass property [8].

A simple algorithm that approximates the greedy path. From the point of view of the motion of the packets, packets routed by algorithm A1, start

moving into their rows and continuously turn right so that they move around their destination row, until they reach it (see Figure 1).

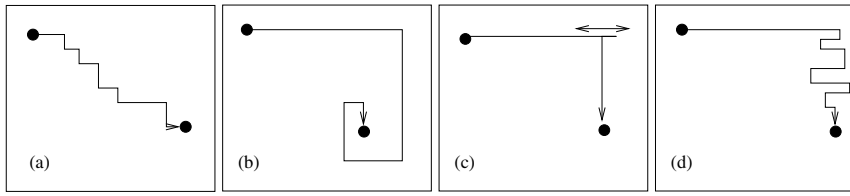


Fig. 1. Typical movement of packets performed by (a) the greedy algorithm, (b) algorithm A1, (c) algorithm KKR, and (d) algorithm A2.

The algorithm KKR [9]. Each packet p starts moving along its row, following the shortest path towards its destination column. When it reaches its destination column, p attempts to enter the column and moves towards its destination. If it fails, it moves "back and forth" until it succesfully turns into the right column.

A variation of the algorithm KKR. Algorithm A2 is based on the following idea: during the time steps that a packet p is moving "back and forth" along its row trying to turn into the correct column, it could also try to turn at some other node trying to decrease the vertical distance of the packet from its destination.

This can be seen as a movement of a packet in its row until it reaches its destination column, and then, greedy movement to the destination processor. Thus, algorithm A2 maintains both properties: it is greedy and also approximates the greedy path.

The typical shape of the paths traversed by the packets during the execution of the algorithms is depicted in Figure 1.

3 Experimentation

The four algorithms where implemented in C, and the results were conducted by simulation experiments on a Pentium III/500Mhz running Solaris 7.

In the static model, packets are initially stored at the injection buffer and start moving according to the routing algorithm. Initially, each processor has one packet stored in its injection buffer. We consider routing problems with *random destinations* (i.e., each packet is assigned as destination a processor, selected among all the processors of the network, uniformly at random) and *random permutations* (i.e., the routing problem is selected uniformly at random among all possible permutations).

In our experiments, the parameter of interest was the running time of the algorithms. Statistics on the running time of the algorithms on routing problems with random destinations are depicted in Table 1. The results for random permutations are close to those for random destinations.

	200×200			500×500		
	Average	Max.	Std dev.	Average	Max.	Std dev.
Greedy	202.69	207	1.509	505.85	512	2.032
A1	201.39	205	1.214	501.16	506	1.412
KKR	205.19	214	2.881	505.13	511	2.553
A2	204.98	217	3.387	505.2	515	2.785

Table 1. Statistics from the execution of the four algorithms on 100 routing problems with random destinations in tori 200×200 and 500×500. The average and maximum observed running time, as well as the standard deviation of the running time is shown.

The performance of all algorithms is very close to the optimal. We believe that all the four algorithms route (almost all) batch routing problems in time $n + O(\log n)$ on the $n \times n$ torus. Such a strong theoretical result has only been proved for algorithm KKR in [9]. Algorithm A1 has slightly better performance than the other three algorithms. Surprisingly, algorithm A2 does not improve the running time of algorithm KKR.

In the dynamic model, packets with random destinations are continuously generated at each processor with a rate λ (i.e., at each time step, a processor generates a packet with probability λ, independently from the other processors) and stored in the injection buffer. Injection buffers have been implemented as FIFO (first–in–first–out) queues; so the network together with the injection buffers is considered as a queueing system. Once a packet leaves the injection buffer of its origin processor, it starts moving according to the routing algorithm until it reaches its destination, where it is stored in the delivery buffer and absorbed (leaves the system).

In our experiments under the dynamic model, parameters of interest were the delay of packets, the number of packets in the system (i.e., the number of packets in injection buffers and packets being routed) and the network throughput, i.e., the maximum injection rate for which the system is stable.

A theoretical maximum value for the maximum injection rate on the $n \times n$ torus is given by $\lambda_{max} = 8/n$ [11]. We alternatively express the injection rate (and the network throughput) as a percentage of its theoretical maximum value. Although we never observe stable behavior of the system for injection rates very close to the theoretical maximum, we observed network throughput up to $72,5\%$.

We performed experiments on the 200×200 torus for injection rates smaller than 50% (see Table 2). In this case, for the four algorithms we study, the network is stable. We observe that, even for small injection rate, packets experience significant delays when routed with the greedy algorithm, while the average number of the packets in the network is large. Especially the average (total) size of injection buffers when the greedy algorithm is used is more than twice the average size of injection buffers when any of the other three algorithms is used.

In our experiments on the 200×200 torus, the network throughput observed was about $0,0276$ (69%) for the greedy algorithm, 0.0254 (63.5%) for algorithm A1 0.0265 (66.25%) for algorithm KKR, and 0.029 (72.5%) for algorithm A2.

	Average delay			Average size of injection buffers		
	$\lambda = 0.005$	$\lambda = 0.01$	$\lambda = 0.02$	$\lambda = 0.005$	$\lambda = 0.01$	$\lambda = 0.02$
Greedy	0.76	2.547	10.932	189	1,026	9,088
A1	0.336	0.943	4.664	156	386	3,835
KKR	0.475	1.304	6.456	78	543	5,383
A2	0.444	0.88	4.444	46	486	3,720

Table 2. Average delay of packets and average total size of injection buffers. The average on the delay was taken among all packets that reached their destinations within $2,500$ steps of execution. The average size of injection buffers was computed for the steps of execution 1000–2500.

References

1. P. Baran. On Distributed Communication Networks. *IEEE Transactions on Communications*, pp. 1–9, 1964.
2. A. Bar–Noy, P. Raghavan, B. Shieber, and H. Tamaki. Fast Deflection Routing for Packets and Worms. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pp. 75–86, 1993.
3. I. Ben–Aroya, T. Eilam, and Schuster. Greedy Hot–Potato Routing on the Two–Dimensional Mesh. *Distributed Computing*, 9(1):3–19, 1995.
4. A. Borodin and J. Hopcroft. Routing, Merging, and Sorting on Parallel Models of Computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.
5. A. Broder and E. Upfal. Dynamic Deflection Routing on Arrays. In *Proc. of the 28th Annual ACM Symposium on the Theory of Computing*, pp. 348–358, 1996.
6. C. Busch, M. Herlihy, and R. Wattenhofer. Randomized Greedy Hot–Potato Routing. In *Proc. of the 11th Annual ACM/SIAM Symposium on Discrete Algorithms (SODA '00)*, pp. 458–466, 2000.
7. U. Feige and P. Raghavan. Exact Analysis of Hot–Potato Routing. In *Proc. of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pp. 553–562, 1992.
8. B. Hajek. Bounds on Evacuation Time for Deflection Routing. *Distributed Computing*, 5:1–6, 1991.
9. C. Kaklamanis, D. Krizanc, and S. Rao. Hot–Potato Routing on Processor Arrays. In *Proc. of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 273–282, 1993.
10. M. Kaufmann, H. Lauer, and H. Schroder. Fast Deterministic Hot–Potato Routing on Meshes. In *Proc. of the 5th International Symposium on Algorithms and Computation*, LNCS 834, Springer–Verlag, pp. 333-341, 1994.
11. F.T. Leighton. Average Case Analysis of Greedy Routing Algorithm on Arrays. In *Proc. of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 2–10, 1990.
12. I. Newman and A. Schuster. Hot–Potato Algorithms for Permutation Routing. *IEEE Transactions on Parallel and Distributed Systems*, 6(11): 1168–1176, 1995.
13. R. Prager. An Algorithm for Routing in Hypercube Networks. Master's thesis, University of Toronto, 1986.

Improving the Up*/Down* Routing Scheme for Networks of Workstations ⋆

José Carlos Sancho and Antonio Robles

Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
P.O.B. 22012,46071 - Valencia, SPAIN
{jcsancho,arobles}@gap.upv.es

Abstract. Networks of workstations (NOWs) are being considered as a cost-effective alternative to parallel computers. Many NOWs are arranged as a switch-based network with irregular topology, which makes routing and deadlock avoidance quite complicated. Current proposals use the $up^*/down^*$ routing algorithm to remove cyclic dependencies between channels and avoid deadlock. Recently, a simple and effective methodology to compute $up^*/down^*$ routing tables has been proposed by us. The resulting $up^*/down^*$ routing scheme makes use of a different link direction assignment to compute routing tables. Assignment of link direction is based on generating an underlying acyclic connected graph from the network graph. In this paper, we propose and evaluate new heuristic rules to compute the underlying graph. Moreover, we propose a traffic balancing algorithm to obtain more efficient $up^*/down^*$ routing tables when source routing is used. Evaluation results show that the routing algorithm based on the new methodology increases throughput by a factor of up to 2.8 in large networks, also reducing latency significantly.

Keywords: Networks of workstations, irregular topologies, routing algorithms, deadlock avoidance.

1 Introduction

NOWs are arranged as a switch-based network with irregular topology which provides the wiring flexibility, scalability, and incremental expansion capability required in this environment. Routing in irregular topologies can be based on either source or distributed routing. In the former case, routing tables are used at each host to obtain the port sequence to be used at intermediate switches to reach the destination. In order to achieve high bandwidth and low latencies, NOWs are often connected using gigabit local area network technologies. There are recent proposals for NOW interconnects like Autonet [8], Myrinet [1], Servernet II [4], and Gigabit Ethernet [9].

Several deadlock-free routing algorithms have been proposed for NOWs, such as $up^*/down^*$ routing [8], adaptive-trail routing [5], minimal adaptive routing [7], and smart-routing [2]. However, we will focus on $up^*/down^*$ routing because it is the most popular routing scheme currently used in commercial networks, like Myrinet [1].

⋆ This work was supported by the Spanish CICYT under Grant TIC97-0897-C04-01.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 882–889, 2000.
© Springer-Verlag Berlin Heidelberg 2000

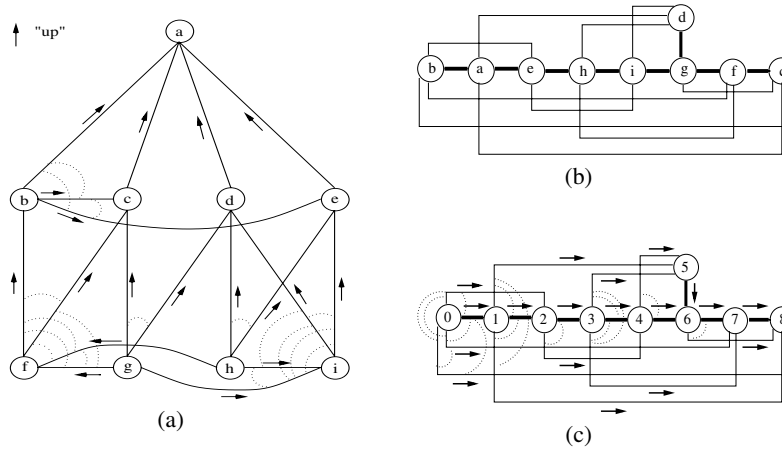


Fig. 1. (a) Generated BFS spanning tree for a 9-switch network with assignment of direction to links. (b) Generated DFS spanning tree, and (c) assignment of direction to links for the same 9-switch network.

In this paper, we propose and evaluate new heuristic rules and a new traffic balancing algorithm that improve the methods to compute $up^*/down^*$ routing tables in a NOW environment when source routing is used. Evaluation results show that the routing algorithm based on the new methodology increases throughput by a factor of up to 2.8 for large networks, also reducing latency significantly. The rest of the paper is organized as follows. In Section 2, the $up^*/down^*$ routing scheme and the methodologies to compute its routing tables are described. In Section 3, new heuristic rules to compute the $up^*/down^*$ routing scheme are proposed. Section 4 describes the proposed traffic balancing algorithm when using source routing. Section 5 shows performance evaluation results. Finally, in Section 6 some conclusions are drawn.

2 Up*/Down* Routing

$Up^*/down^*$ is the most popular routing scheme currently used in commercial networks. In order to compute $up^*/down^*$ routing tables, different methodologies can be applied. These methodologies are based on an assignment of direction ("up" or "down") to the operational links in the network by building a spanning tree. These methodologies differ in the type of graph to be built. One methodology is based on a BFS spanning tree, such as it was proposed in Autonet [8], whereas another methodology is based on a DFS spanning tree, as it has been recently proposed in [6].

In networks without virtual channels, the only practical way of avoiding deadlocks consists of restricting routing in such a way that cyclic channel dependencies [1] are

[1] There is a channel dependency from a channel c_i to a channel c_j if a message can hold c_i and request c_j. In other words, the routing algorithm allows the use of c_j after reserving c_i. Also, there is a routing restriction when there is no channel denpendency.

avoided [3]. To avoid deadlocks while still allowing all links to be used, $up^*/down^*$ routing uses the following rule: a legal route must traverse zero or more links in "up" direction followed by zero or more links in "down" direction. Thus, cyclic channel dependencies are avoided by imposing routing restrictions, because a message cannot traverse a link along the "up" direction after having traversed one in "down" direction. Next, we describe how to compute both a BFS and DFS spanning tree, and how to assign direction to links in each graph.

2.1 Computing a BFS Spanning Tree

First, to compute a BFS spanning tree, a switch must be chosen as the root. Starting from the root, the rest of the switches in the network are arranged on a single BFS spanning tree. The "up" end of each link is defined as: $1)$ the end whose switch is closer to the root in the spanning tree; $2)$ the end whose switch has the lower identifier, if both ends are at switches at the same tree level. Figure 1(a) shows the resulting link direction assignment for a 9-switch network.

2.2 Computing a DFS Spanning Tree

Like in BFS spanning trees, an initial switch must be chosen as the root before starting the computation of a DFS spanning tree. The rest of the switches are added following a recursive procedure [6]. This procedure builds a path that connects all the switches in the network. Figure 1(b) shows the DFS spanning tree obtained from the same network graph used in Figure 1(a). Unlike in the BFS spanning tree, adding switches to build the path is made by using heuristic rules. We will address this issue later.

Next, before assigning direction to links, switches in the network must be labeled with positive integer numbers. A different label is assigned to each switch. The "up" end of each link is defined as the end whose switch has the higher label. Figure 1(c) shows the label assigned to each switch. Note that, the DFS spanning tree achieves a lower number of routing restrictions, as can be seen by dashed lines in Figures 1(a) and 1(c) for the BFS and DFS spanning trees, respectively.

3 Applying New Heuristic Rules

Several spanning trees can be computed. In order to achieve better performance, heuristics to find the suitable spanning tree are needed. For BFS spanning trees, heuristic rules can only be applied to choose the root switch. The number of different BFS spanning trees that can be computed on a network graph is limited by the number of switches in the network. However, when computing a DFS spanning tree, heuristic rules can be applied to both the selection of the root switch and the selection of the following switches of the spanning tree. Notice that the number of spanning trees that could be computed in this case is very large.

We first focus on the heuristic rules for selecting the root switch. So far, two approaches have been used to select the root of a spanning tree: (R0) to select the switch with identifier equal to zero, like in DEC AN1 [8]; (R1) to select the switch with lower

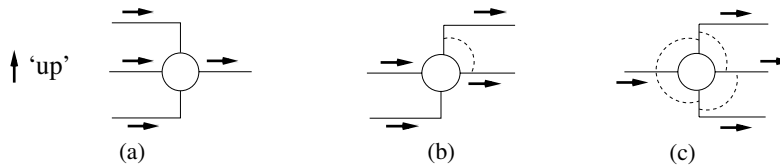


Fig. 2. Different link orientation patterns in a DFS spanning tree.

average topological distance to the rest of the switches, like Myrinet [1]. We propose a new heuristic rule that will be referred to as R2. The heuristic is based on computing all the spanning trees and selecting one of them based on two behavioral routing metrics, that is: (1) the average number of links in the shortest routing paths between hosts over all pairs of hosts, referred to as *average distance*; and (2) the maximum number of routing paths crossing through any network channel, referred to as *crossing paths*. We first compute the metrics for each spanning tree obtained by selecting the root among every switch in the network. Finally, the switch selected as the root will be the one that provides the lower value for the crossing paths metric. In case of tie, the switch with lower value of average distance will be selected. In short, the switch selected as the root will be the one that allows more messages to follow minimal paths and provides a better traffic balancing. The time complexity to compute the new heuristic rule is $O(n^3)$, where n is the number of switches.

Unlike BFS spanning tree, after selecting the root switch, a DFS spanning tree still allows heuristic rules to be applied to the rest of the switches when building the spanning tree. We propose the two following heuristic rules: (H1) The switch with higher average topological distance to the rest of the switches is selected as the next switch in the spanning tree, and so on. This heuristic was proposed in [6]. (H2) The switch with a higher number of links connecting to switches that already belong to the spanning tree is selected as the next switch. In case of tie, the H1 heuristic rule is applied. The H2 heuristic reduces the number of routing restrictions by increasing the number of switches whose links exhibit the orientation patterns shown in the Figures 2(a) and 2(b), which provide a lower number of routing restrictions in the switch than that provided by the link orientation pattern shown in Figure 2(c).

Table 1 shows the values of the behavioral routing metrics computed for several network[2]sizes using the $up^*/down^*$ routing algorithm based on both BFS and DFS spanning trees, which have been obtained according to the heuristic rules proposed above[3]. Besides the *average distance* and the *crossing path* metrics we have also included the *restrictions per switch* metric that is the average number of routing restrictions per switch. As can be seen, a lower values of metrics are obtained when using the R2 and H2 heuristic rules to compute spanning trees.

[2] For further details on topology generation, see Section 5.1.

[3] For DFS spanning trees, we assume that the R2 heuristic is used to select the root.

Table 1. Behavioral routing metrics for BFS and DFS spanning trees using different heuristics.

Spanning tree	Network size	Average distance		Crossing paths		Restrictions per node	
		R1	R2	R1	R2	R1	R2
BFS	16	2.208	2.133	37	23	3.375	3.125
	32	3.102	2.871	173	63	3.562	2.937
	64	4.013	3.787	593	238	3.281	2.875
		H1	H2	H1	H2	H1	H2
DFS	16	2.108	2.091	23	23	3.125	2.875
	32	2.792	2.752	73	43	2.821	2.625
	64	3.634	3.590	204	190	2.687	2.585

4 Traf c Balancing Algorithm

When a routing algorithm able to provide partial adaptivity, such as $up^*/down^*$ routing, is implemented using source routing, a strategy to select a single path between each pair of hosts is needed. Different selection policies can be applied, such as random and round-robin selections. However, they do not guarantee a suitable traffic balancing in the network, which may reduce network performance. We propose a traffic balancig algorithm that tries to achieve an uniform channel utilization, avoiding that a few channels become a bottleneck in the network.

First, the algorithm associates a counter to every channel in the network. Each counter is initialized to the number of routing paths crossing the channel, that is, the channel utilization. Additionally, a cost function associated to every routing path according to its channel utilization is evaluated. The procedure defined below is applied repetitively to the channel with highest value of counter. In each step, a routing path crossing the channel is selected to be removed if there is more than one routing path between the source and the destination switch of this routing path. In this way, we prevent the network to become disconnected. When a routing path is removed, the counters associated with every channel crossed by the path are updated. If there is more than one routing path able to be removed in a channel, the algorithm will first choose the routing path whose source and destination hosts have the highest number of routing paths between them. The algorithm finishes when the number of routing paths between every pair of hosts is reduced down to the unit. The time complexity to compute this traffic balancing algorithm is $O(n^2 * diameter)$, where n is the number of switches. This time is much lower than that exhibited by other proposals, such as smart-routing [2].

5 Performance Evaluation

In this section, we evaluate by simulation the performance of the $up^*/down^*$ routing scheme when the heuristic rules and the traffic balancing algorithm proposed in Sections 3 and 4, respectively, are applied to compute the routing tables. Table 2 shows

the acronyms for the $up^*/down^*$ routing algorithms evaluated according to type of spanning tree, heuristic, and traffic balancing algorithm used.

Table 2. Acronyms used for the $up^*/down^*$ routing algorithms.

Routing algorithm	Spanning tree	Root heuristic	Path heuristic	Traffic balancing
UD_BFS1	BFS	R1	-	No
UD_BFS2	BFS	R2	-	No
UD_BFS2b	BFS	R2	-	Yes
UD_DFS1	DFS	R2	H1	No
UD_DFS2	DFS	R2	H2	No
UD_DFS2b	DFS	R2	H2	Yes

5.1 Network Model

Network topology is completely irregular and has been generated randomly. We have evaluated networks with 16, 32, and 64 switches. For space reasons, the results for 64 switches have not been plotted. We have generated ten different topologies for each network size analyzed. The maximum variation in throughput improvement of UD_DFS2b routing with respect to UD_BFS1 routing is not larger than 20%. Results plotted in this paper correspond to the topologies that achieve the average behavior for each network size. We assume that every switch in the network has 8 ports, using 4 ports to connect to workstations and leaving 4 ports to connect to other switches. For message length, 32-flit and 512-flit messages were considered. Different message destination distributions have been used like uniform, bit-reversal, and matrix transpose.

In order to obtain realistic simulation results, we have used timing parameters for the switches taken from a commercial network. We have selected Myrinet because it is becoming increasingly popular due to having very good performance/cost ratio. According to Myrinet switches, the latency through the switch for the first flit is 150 ns, and after transmitting the first flit, the switch transfers at the link rate of 6.25 ns per flit. The clock cycle is 6.25 ns. Each switch has a crossbar whose arbiter processes one message header at a time. Flits are one byte wide and the physical channel is one flit wide. Also, source routing and wormhole switching is used like in Myrinet.

5.2 Simulation Results

Figures 3(a) and 3(b) show the average message latency versus accepted traffic for networks with 16 and 32 switches, respectively. Message size is 32 flits and uniform destination distribution is used. We can observe that the new heuristic (H2) to compute the DFS spanning tree allows UD_DFS2 to reduce latency with respect to UD_DFS1

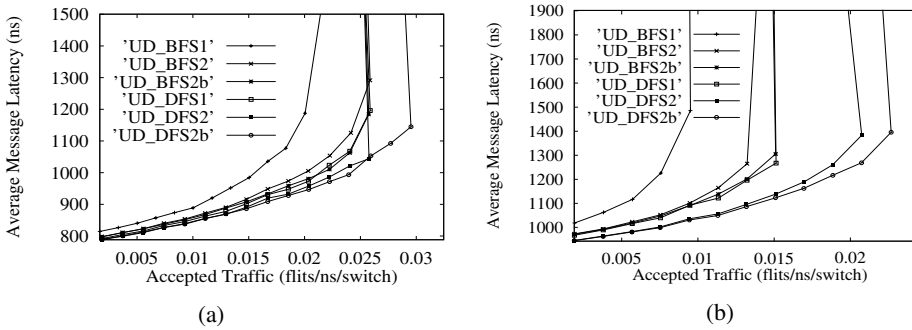


Fig. 3. Average message latency vs accepted traffic. Network size is (a) 16 and (b) 32 switches. Message length is 32 flits. Uniform distribution.

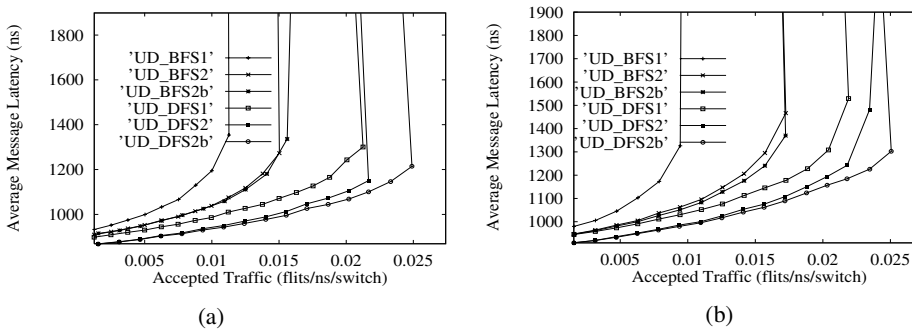


Fig. 4. Average message latency vs accepted traffic. Network size is 32 switches. Message length is 32 flits. (a) Bit-reversal and (b) matrix transpose message distributions.

for every value of traffic. It is due to the fact that the new heuristic introduces lower number of routing restrictions than the previous heuristic (H1), allowing more messages to follow minimal paths. Obviously, the improvement is higher in large networks because messages can profit more from following minimal paths. The improvement in throughput of UD_DFS2 with respect to UD_BFS1 achieves a factor of up to 2.8 for a 64-switch network. Also, the new heuristic (R2) to select the root significantly improves the performance of the $up^*/down^*$ routing scheme based on BFS spanning tree with respect to the R1 heuristic. The improvement in throughput of UD_BFS2 with respect to UD_BFS1 ranges from 20% for small networks to 60% for large networks.

Moreover, the traffic balancing algorithm only contributes to improve throughput for small network sizes, especially when $up^*/down^*$ routing is based on DFS spanning tree. It is due to the fact that channel utilization is higher than for large networks. As a consequence, an algorithm to balance traffic will achieve more benefits. The improvement in throughput of UD_DFS2b with respect to UD_DFS2 is about 16% in 16-switch networks.

For space reasons, the results for long message (512-flits) are not plotted. The improvement in performance of the $up^*/down^*$ routing schemes based on a DFS spanning tree with respect to those based on BFS spanning tree decreases slightly with respect to the one achieved with short messages. Similar results were obtained in [6].

Figures 4(a) and 4(b) show the results for a 32-switch network when using message distributions with temporal locality, such as bit-reversal and matrix transpose. The improvement in performance of UD_DFS2 with respect to UD_DFS1 is noticeably decreased, although the latency reduction is still significant for the entire range of traffic. Notice that UD_DFS2b increases throughput with respect to UD_BFS1 up to a factor of 2.5.

6 Conclusions

In this paper, we have proposed new heuristics to obtain the underlying graph used by the $up^*/down^*$ routing scheme to compute the routing tables. Moreover, an algorithm to balance the traffic in the network using source routing has been proposed in order to avoid that some channels become a bottleneck in the network. The main contribution of these techniques is that they are able to improve network performance without adding resources to the network that would increase its cost. Simply, routing tables have to be updated.

The simulation results modeling a Myrinet network show that the $up^*/down^*$ routing algorithm based on DFS spanning tree, when the new heuristic to compute the spanning tree and the traffic balancing algorithm are applied, almost triples the throughput in large networks with respect to the $up^*/down^*$ routing algorithm based on BFS spanning tree currently used in commercial networks. For smaller networks, performance improvement is also smaller but the proposed heuristic rules always improve latency and throughput.

References

1. N. J. Boden et al., Myrinet - A gigabit per second local area network, *IEEE Micro*, vol. 15, Feb. 1995.
2. L. Cherkasova, V. Kotov, and T. Rockicki, Fibre channel fabrics: Evaluation and design, *29th Hawaii International Conference on System Sciences*, Feb. 1995.
3. W. J. Dally and C. L. Seitz, Deadlock-free message routing in multiprocessors interconnection networks, *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547-553, May. 1987.
4. D. García and W. Watson, Servernet II, in *Proceedings of the 1997 Parallel Computer, Routing, and Communication Workshop*, Jun 1997.
5. W. Qiao and L. M. Ni., Adaptive routing in irregular networks using cut-trough switches, in *Proc. of the 1996 Int. Conf. on Parallel Processing*, Aug. 1996.
6. J.C. Sancho, A. Robles, and J. Duato, New Methodology to Compute Deadlock-Free Routing Tables for Irregular Networks, in *in Proc. of CANPC'00*, Jan. 2000.
7. F. Silla and J. Duato, Improving the Efficiency of Adaptive Routing in Networks with Irregular Topology, in *1997 Int. Conference on High Performance Computing*, Dec. 1997.
8. M. D. Schroeder et al., Autonet: A high-speed, self-configuring local area network using point-to-point links, *SRC research report 59*, DEC, Apr. 1990.
9. R. Sheifert, *Gigabit Ethernet*, Addison-Wesley, April 1998.

Deadlock Avoidance for Wormhole Based Switches

Ingebjørg Theiss and Olav Lysne

Department of Informatics, University of Oslo
theiss@ifi.uio.no, olavly@ifi.uio.no

Abstract This paper considers the architecture of switches. In particular we study how virtual cut-through and wormhole networks can be used as the switch internal interconnect. Introducing such switches into a deadlock free interconnect may give rise to new deadlocks. Previously, to reason that no deadlocks are created, the resulting system was considered globally, that is, the interconnects of the switches themselves was considered as part of the system. Using flow control across the switch will eliminate the possibility of creating new deadlocks, and further global reasoning will not be necessary.

1 Introduction

Wormhole routing (WHR) has recently been brought into new applications, such as high speed LANs and SANs (i.e. Autonet [1], Myrinet [2] and Servernet [3]), and has been proposed as the internal fabric of switches [4, 5]. The latter is interesting for highly scalable switches. One could consider using a k-ary n-cube [6] as the internal topology when building such a switch, but the result will have an unfortunate property. Unlike a simple crossbar router, the WHR switch may block, that is, a packet may have to wait for another packet, even when the packets have different source and destination ports. Using other internal topologies will generally not change this situation.

The problem addressed is that of new deadlocks appearing when crossbar routers are exchanged with blocking switches in a given configuration. This problem is previously mentioned in [5] where it is emphasized that nonblocking switches help preserve inter-switch deadlock freedom, while blocking switches are good when only intra-switch deadlock freedom is required. In [7], however, Lysne argues that if blocking switches are used, reconsidering the structure of the switch-internal configurations and adding a few virtual channels to avoid what he calls aggregated dependencies, will ensure that the blocking property cannot cause inter-switch deadlocks.

Another approach is to use extra logic in the switch to avoid the blocking property. This paper considers the latter approach, where the extra logic used is a flow control protocol across the switch.

In [8] we considered using a wormhole routed switch in an SCI configuration [9], and addressed the problems connected to the non-blocking property of the

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 890–899, 2000.
© Springer-Verlag Berlin Heidelberg 2000

switch. The methods in this paper are not restricted to SCI. The solutions found in [8] are generalized to technologies where a switch cannot drop packets. Also, a substantial amount of simulations have been made: three routing techniques, two topologies and several traffic patterns have been tested.

The outline of the following sections is as follows: First, the deadlock problem arising with blocking switches is introduced. Next, three different flow control methods removing the blocking property and thereby the deadlock are proposed. The performance of these methods are then analyzed through simulation, and finally the scalability issues are discussed, showing that very large switches can be constructed with limited buffer capacity at each port.

2 Deadlock Caused by Blocking Switches

To simplify our discussion, we introduce some definitions. We use the term *chip* to imply that the node under discussion is a simple one-chip routing node, such as a crossbar router. A *switch* is understood to be a more complex routing node, built with an internal topology of chips connected with links. Typically, a chip has 4 to 32 ports, while a switch is virtually unlimited.

When a node lets data be transferred between any pair of ports independently of traffic from any other input port, as long as the output port is not occupied, and every output port also fairly arbitrates between input ports, the node is non-blocking and fair (*NBF*). In the following, a chip denotes an NBF-chip.

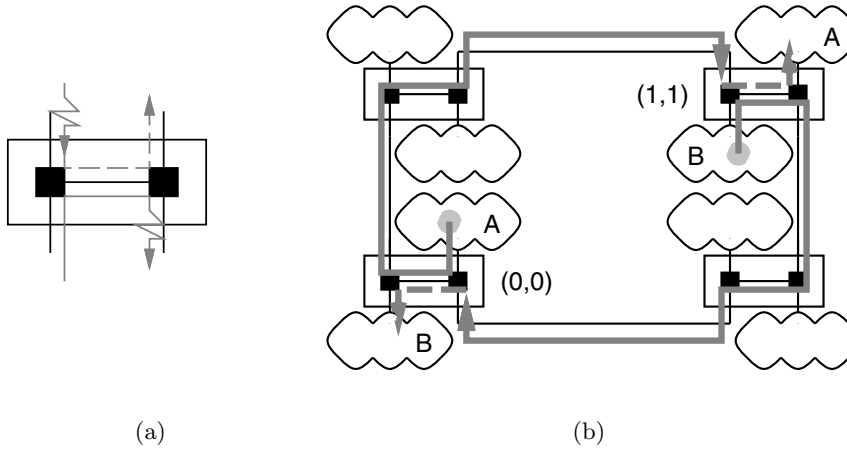


Fig.1. (a) A blocking switch (b) Deadlock in a compound configuration

As an example of a non-NBF-node, consider a switch with 4 ports and a full duplex 2-ary 1-cube as the WHR internal topology connecting the ports

(Fig 1 (a)). Each NBF-chip (the black squares in the figure) in this topology is connected to two ports. If both inputs at one NBF-chip wishes to send packets to the two outputs at the other NBF-chip, they are competing for the inter-chip link, even though the destinations are distinct. If the input packet which wins the competition is blocked at its output, the packet from the other input is also blocked.

A distinction between *internal* and *external configurations* in configurations using switches as routing nodes is also necessary. The former denotes the internals of the switches, the latter expresses the configuration disregarding the internal configurations, as if the routing nodes were chips. Furthermore, a *compound configuration* regards the entire structure, both the external and the internal configurations. A compound configuration is *externally deadlock free* when its external configuration is deadlock free, and *internally deadlock free* when all its internal configurations are deadlock free.

Now, consider a structure with a deadlock free external configuration. It is obvious that if the internal configurations of the switches have cyclic dependencies, the compound configuration also has cyclic dependencies. However, even if the internal configurations are deadlock free, the compound configuration may still deadlock. Inserting non-NBF deadlock free switches into an externally deadlock free configuration can induce cyclic dependencies in the compound configuration.

Let our external configuration be a wormhole routed 2-ary 2-cube using dimension order routing. We use the WHR 2-ary 1-cube from Fig. 1 (a) as the internal configurations. The compound configuration is both externally and internally deadlock free, the configuration is illustrated in Fig. 1 (b).

As an example of a deadlocked situation in this configuration, consider two packets, from A to A' and from B to B'. The packets are initiated simultaneously. The packet from A fills all resources on its path to switch (1,1), thereby occupying the horizontal interconnect of switch (0,0). A similar situation is caused by the packet from B. Since the horizontal interconnect in (1,1) is occupied by the packet from B, the packet from A is blocked. The packet from B is in its turn blocked, waiting for the horizontal interconnect in (0,0) occupied by the packet from A. A circular wait is present, and the configuration is deadlocked.

This problem is not limited to wormhole routed networks, it applies to both Store And Forward (SAF) and Virtual Cut-Through (VCT). This follows easily from the fact that any deadlock situation in WHR can be seen as a deadlock situation in both SAF and VCT by considering the WHR flits as SAF and VCT packets. The deadlock can also be found when combining these technologies, ie. a SAF external configuration and WHR internal configurations.

To avoid inducing a cyclic dependency we give the switches sufficient properties to become NBF, allowing us to disregard the internal configurations. To achieve this, three requirements must be imposed: *(i)* the switches must be deadlock free, *(ii)* a blocked stream crossing a switch from port A to port B cannot persistently block packets crossing the switch from C to D (when C and D are disjoint from A and B, respectively), *(iii)* the packet scheduling within the switch must be fair.

Theorem 1. *If we insert switches that fulfill requirements (i), (ii) and (iii) into an externally deadlock free configuration, the compound configuration will also be deadlock free.*

Proof 1. The actual proof is by contradiction – we assume our switches introduce a cyclic dependency which has lead to a deadlock situation, and show that this gives us a contradiction. The contradiction is given by the following steps:

- The introduced deadlock situation must stem from a cycle of dependencies that includes some inserted switches.
- For each of the participating internal configurations the following must hold:
  - There must be an involved packet that cannot be transmitted successfully from one end of the switch to the other. Assume that the packet needs to be transmitted from buffer A to buffer B.
  - The requirements (i), (ii) and (iii) implies that the block is caused by buffer B being eternally full, otherwise there is no cause for blocking.
  - Because this is a deadlock situation there must be a cycle of actions all waiting for each other that includes the successful transmission of our packet across the switch.
  - The only resource that this packet is eternally occupying is space in the buffer where it is situated. Therefore, since we have a deadlock, this buffer must be full.
  - This means that the blocking situation local to this switch is due to full buffers at both interfaces.
- Since the ring of dependencies in the deadlock situation are due to full buffers at each end of all the participating internal configurations, this would have been a deadlock with NBF-chips as well. This contradicts the premise of an externally deadlock free configuration. ■

To fulfill requirements *(i)*, *(ii)* and *(iii)*, we shall look at switches with deadlock free WHR configurations. The switch outputs arbitrate fairly between inputs, and in addition, the switches have flow control between inputs and outputs. The flow control will assure that when a packet is allowed to cross the switch, it is in fact removed from the switch's internal network, allowing other packets to other destinations to use the network. From our example in Fig. 1 (b), the flow control will always free the shared horizontal interconnects, and these interconnects will be fairly shared by the two streams.

3 Flow Control Methods

In this section several methods for implementing the properties identified in the previous section are discussed. All methods uses well-known techniques to control the flow across the interconnect in such a way that the non-blocking and fairness properties are maintained. The performances of the flow control methods are investigated through simulations, and the results are discussed.

3.1 Source Driven Approach

The first flow control method, the source driven approach (SDA), lets the source port initiate communication by sending a data packet to the network, but the port keeps a copy in case retransmission is necessary. After transmitting, the source waits for an acknowledge packet from the destination. The acknowledge packet indicates whether the destination port accepted the data packet or not, the latter may occur if the output port's buffers are full. In case of a positive acknowledge, the source copy can be discarded, and the next data packet waiting can be transmitted. In case of a negative acknowledge, the pending data packet must be retransmitted. This process is repeated until the acknowledge indicates acceptance.

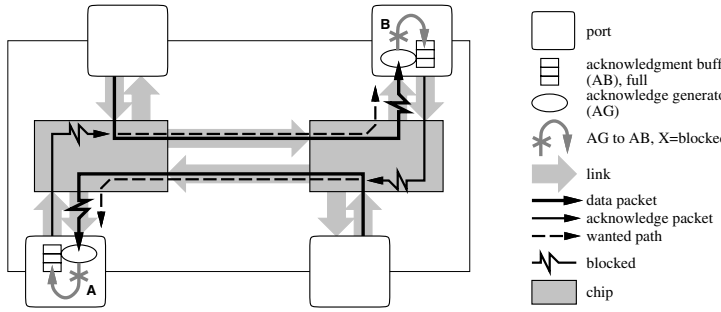


Fig.2. Deadlock in SDA using only one transport medium

The destination port returns an acknowledge packet by putting it in an acknowledge packet buffer, where it will reside until the switch internal network has capacity to transmit it. If this buffer is full, the destination port must leave the data packet in the internal network, blocking a substantial amount of resources there. If acknowledgment traffic uses the same transport medium as data traffic, a deadlock situation may occur. Consider the two ports A and B in Fig. 2. A data packet is going to A through the link connecting B and A. A's acknowledgment buffer is full, so A has to block the data packet until there is room. The corresponding situation applies at B. The acknowledge packets at the top of the acknowledgment buffers are requesting the links occupied by the data packets, and have to wait until the links are freed. There is a circular wait, and the system is deadlocked. Using a dedicated transport network for the acknowledge packets, either as a logical or a physical network, solves this problem.

To ensure fairness, SDA uses an A-B aging scheme when packets are rejected at the destination port [9]. The negative acknowledge informs the source port of how to mark retransmitted packets. From this information, the destination can guarantee any packet a maximum waiting time, that is, an upper bound on the number of other packets being served first.

Letting the source port be idle while waiting for the flow packet is an obvious subject for improvement by allowing the transmitting protocol engine to have M active buffers holding the data packets waiting for acknowledgment. For simplicity, we have used $M = 1$.

3.2 Destination Driven Approach

In the destination driven approach (DDA), the source port has destination associated counters (*credits*) indicating how many packets the source port is allowed to send to a destination port. The credit is decremented when a packet is sent to the destination port, and incremented upon receipt of a special flow control packet returned by this destination port. The destination port accepts all data packets actually sent by the source port.

Since source ports cannot possibly know the credits of its neighbors, each destination must have buffer space dedicated to each source port, and the space must be large enough to store at least one data packet. When a packet in such a dedicated buffer has successfully been delivered to the receiver on the local subnet, the special flow control packet is sent to the source port to increase its credit again. When the maximum value of credits is 1, DDA will degenerate to a stop and wait-protocol.

The required buffer space limits the scalability of DDA to some extent. Using a simple technique, where each source has room for one data packet in each destination, the total buffer space needed is $\#sources \times \#destinations \times maximumpacketsize$ which is of order $O(\#sources^2)$, since a source is normally also a destination. It could be possible to timeshare buffer space by some fair strategy, but this method will not scale well either, neither in time dimension nor in buffer cost.

The internal deadlock problem occuring in SDA is not relevant in DDA. No packet is sent unless it will be accepted by the destination, so all packets will eventually be removed from the internal network, making room for the flow control packets.

Fairness is guaranteed when the selected strategy for submitting packets from the destination buffer onto the local network segment is fair. In our simulations, we use a round robin strategy.

3.3 Draining Network Approach

A third method, the draining network approach (DNA), is a modified version of SDA; DNA needs just a single network. To avoid the deadlock problem SDA solves by using two networks, the destination in DNA guarantees that all packets will be removed from the transport medium upon arrival. Consequently, if no new data packets are sent, the transport medium will eventually be emptied (drained), which implies that no packet can block acknowledge packets eternally.

Some strategy must be used to solve the full acknowledgment buffer problem, otherwise the switch will drop packets. Let each source port in DNA provide

buffer space large enough to keep the highest number of acknowledge packets needed at one time. As for SDA, the number of pending packets at each source is set to 1 ($M = 1$). This means that each destination buffer must have room for *#sources* acknowledgments, and total buffer space needed in the switch is therefore *#sources* $\times$ *#destinations* $\times$ *size of acknowledge packet*. This order is similar to the non-scalable situation of DDA, but we exploit the characteristics of the worst case, and relax the strict FIFO strategy used for flow packets by SDA and DDA, thereby saving a respectable amount of buffer space. Instead of storing the entire acknowledge packet, each destination port holds an array with one index for each source. One array location is as small as 3 bits. A null value in a location indicates that no acknowledge packet is pending for this source port, while the other values can store the status of the latest received packet from this source port. A round robin strategy is used to send the acknowledge packets to the internal network.

In all three approaches, we give acknowledge packets the highest priority, that is, when choosing between sending an acknowledge packet and a data packet on the internal network, we will always send the acknowledge packet, regardless of how long the data packet has been waiting.

3.4 Simulation

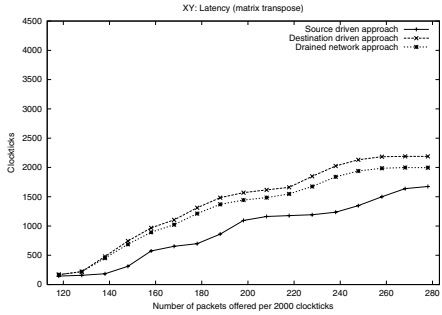
Our simulations of one switch comprise three different configurations, all using WHR. Two topologies are used, a 6×6 2-dimensional grid and a 4×4 Clos network [10].

Using the grid, we have simulated two different routing algorithms, the XY-routing algorithm [11] and the West First algorithm [12]. XY-routing is a deadlock free, dimension ordered, deterministic routing protocol. West First is a partially adaptive, deadlock free routing algorithm based on the Turn Model.

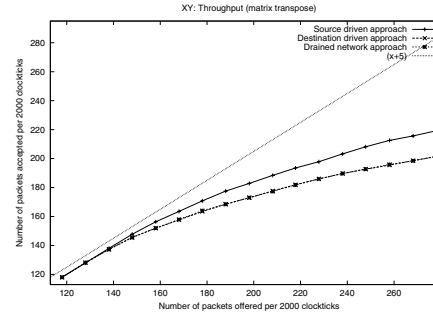
These configurations have been tested with four traffic patterns: uniform, hot-spot, matrix transpose and matrix merge south. The uniform and the hot-spot patterns let sources communicate with different destinations. Each time a packet is transmitted, its destination is randomly picked. As the names suggest, with uniform traffic all destinations have the same probability of being picked, with hot-spot, destinations have a higher chance of being picked the closer they are to the hot-spot. In both matrix patterns, each source port picks a destination at startup, and never sends to anyone else. In matrix transpose each destination port receives packets from one source port only, while in merge south all source ports send southwards, and each destination receives packets from two sources.

The routing algorithm simulated with the Clos network is very simple: all packets are routed adaptively into the middle layer, and then deterministically to their destinations.

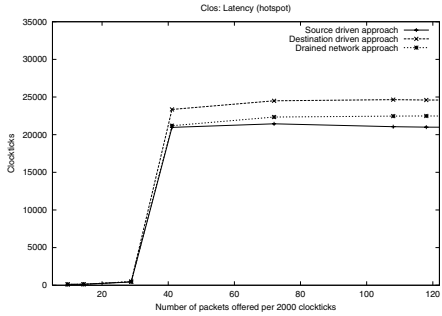
The Clos configuration has also been tested with four traffic patterns, uniform and hot-spot as in the grid, and reflection and split-down, which resembles the matrix patterns in that each source picks one destination, and in the latter, each destination has two sources.



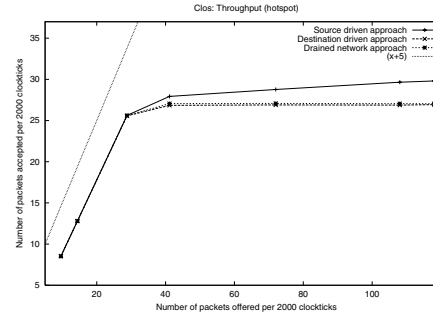
(a) All matrix pattern latencies are similar to this plot.



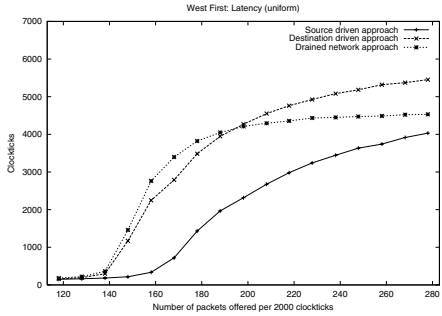
(b) All matrix pattern throughputs are similar to this plot, but scales vary



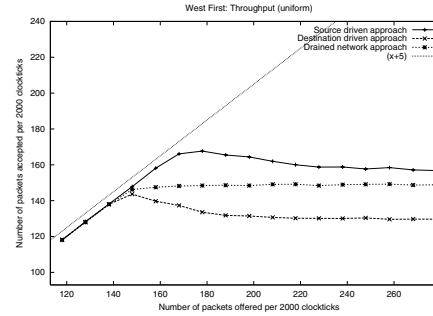
(c) All hot spot latencies are similar, but in West First, SDA crosses DNA



(d) All hot spot throughputs are similar to this plot



(e) Unlike the other uniform latency plots, West First has higher latency and DDA crosses DNA



(f) Unlike the other uniform throughputs, DNA is better than DDA in West First

Fig.3. Selected simulation results

Packets arrive from the network to the source port at uniformly distributed random time intervals. When the average time interval decreases, the network load increases. The graphs below show the latency and throughput of each flow control method with increasing load, the complete set of results can be found in [13]. Packets are 90 flits long.

The most prominent result is that the SDA performs better than both DDA and DNA, regardless of workload, traffic pattern, topology and routing algorithm. (A couple of exceptions: latency is slightly worse in SDA than DNA using West First routing and the hot-spot pattern, and throughput in XY shortly drops below DDA and DNA using the matrix merge south pattern.) With the uniform traffic pattern, SDA is significantly better. Using XY the throughput graphs show that SDA handles traffic up to app. 220 packets per 2000 clock ticks, while MDM and TTM only handle ca. 190, SDM is approximately 15% "better". Using West First SDA is approximately 14% "better", using the Clos network approximately 25%.

Generally, SDA is much better, but using the matrix patterns in XY, hot-spot in West First and reflection for the Clos network the differences are relatively small.

The results from comparing DDA with DNA variates more. Using the uniform traffic pattern, DDA performs better with XY and particularly in the Clos network, while DNA works best with West First. However, even though DNA is logically similar to SDA, it performs very similar to DNA. For uniform traffic, DDA has better throughput than DNA when $M = 1$, but not by an extensive amount. Latency is much better, though. This similarity in performance is possibly due to the fact that the sources send to the same destination. For the uniform traffic, DNA does not have to wait for an acknowledgment from the destination before sending the next packet to a new destination port, since it is credit based. This is no advantage when the destination port is static.

With the grid topology and XY routing, ports at the edge of the network have higher latency and lower throughput than ports in the middle. Using the adaptive routing algorithm, the eastern nodes have a lower service rate. Using the symmetric Clos network, no nodes are disfavoured.

4 Conclusion

We have seen how an end to end flow control method across a switch can hide the blocking property and allow switches to be used as routing nodes in an interconnect such as if they were non-blocking.

There are several ways of how to implement such a flow control, we proposed both source driven and destination driven approaches. By simulation, we found the source driven approach to have better performance than the destination driven approach, but at a cost of an extra internal network. To avoid the extra network, a third approach, also source driven, was suggested, and found to have lower costs than the destination driven approach, but basically the same performance.

Further work in this area could be to add more topologies and routing techniques, or to find out how the configurations behave if the network segments at the output ports becomes saturated.

Another issue is to lower the costs of the drained network approach further. By implementing an additional timeout strategy and allowing a very small probability for dropped packets, the costs could be strongly reduced.

A spin-off of our investigation is the Clos vs. mesh issue. Apparently, our Clos network performs better than our meshes. This is not very surprising, considering that the hop distance in the Clos network is only two. The chances of a header flit wanting an occupied out-link are therefore far less than in a mesh, where the average hop distance is much higher. The incentive for using wormhole routing in Clos is for the same reason less, though. On the cost issue; our small Clos network is more cost effective when counting chips and links, but it doesn't scale as well as mesh networks do.

References

[1] M. D. Schroder et.al. Autonet: a high-speed, self-configuring local area network using point-to-point links. SRC Research Report 59, Digital Equipment Corporation, 1990.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[3] Robert W. Horst and David Garcia. ServerNet SAN I/O Architecture. *Hot Interconnects V*, 1997.

[4] Vibhavasu Vuppala and Lionel M. Ni. Design of A Scalable IP Router. In *Hot Interconnects V*, 1997.

[5] Lionel M. Ni, Wenjian Qiao, and Mingyao Yang. Switches and Switch Interconnects. In *Proceedings of the Fourth International Conference on Massively Parallel Processing Using Optical Interconnections*, pages 122–129, 1997.

[6] José Duato, Sudhakar Yalamanchili, and Lionel M. Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society, 1997.

[7] Olav Lysne. Deadlock Avoidance for Switches based on Wormhole Networks. In *Proceedings of the 1999 International Conference on Parallel Processing, Aizu-Wakamatsu (Japan)*, pages 68–74. IEEE Computer Society Press, 1999. ISBN 0-7695-0350-0.

[8] Geir Horn, Ingebjørg Theiss, Olav Lysne, and Tor Skeie. Switched SCI Systems. In *Scalable Coherent Interface: Technology and Applications*, Proceedings of SCI Europe '98, pages 13–24. Cheshire Henbury, 1998.

[9] IEEE 1596-1992. The Scalable Coherent Interface (SCI), 1992.

[10] Charles Clos. A Study of Non-Blocking Switching Networks. *Bell Syst. Tech. J.*, 32:406–424, 1953.

[11] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *Computer*, 1993.

[12] Christopher J. Glass and Lionel M. Ni. The Turn Model for Adaptive Routing. *Journal of the Association for Computing Machinery*, 1994.

[13] Ingebjørg Theiss and Olav Lysne. Simulation results for deadlock free wormhole based switches. Research Report 284, University of Oslo, Department. of Informatics, June 2000. ISBN 82-7368-232-3 ISSN 0806-3036.

An Analytical Model of Adaptive Wormhole Routing with Deadlock Recovery

Mohamed Ould-Khaoua and Ahmad Khonsari

Computer Science Department, Strathclyde University, Glasgow, U.K.
{mohamed, ak}@cs.strath.ac.uk

Abstract: This paper proposes a new analytical model to predict the mean message latency in *k*-ary *n*-cubes with Compressionless routing, a recovery-based fully-adaptive routing proposed by Kim *et al* [2].

1 Introduction

Deadlock recovery as a viable alternative to deadlock avoidance has recently gained consideration in the scientific community. It has been shown that deadlocks are quite rare except when the network is close to saturation [2]. Thus the hardware dedicated for deadlock avoidance is not necessary most of the time. This consideration has motivated the authors in [2] to introduce Compressionless Routing as a framework for developing recovery-based fully-adaptive routing algorithms. This paper proposes the first analytical model of compressionless routing in *k*-ary *n*-cubes.

2 The Analytical Model

Compresionless routing and the router structure are described in detail in [2]. The model is based on the following assumptions [3].

i) Message destinations are uniformly distributed. Nodes generate traffic independently of each other, following a Poisson process with a mean rate of λ_g messages/cycle. The message length is *m* flits, where *m* is a random variable with a mean $\overline{m}$. A flit requires one-cycle transmission time across a physical channel.

ii) The timeout period is τ cycles. The probability of timeout at a channel is independent of the subsequent channels. When a transmission failure occurs due to timeout, a message is transmitted after a time gap of *V* cycles, where *V* may be any random variable with a mean $\overline{V}$. Successive message re-transmissions are independent of the each other.

iii) L $(L \geq 1)$ virtual channels are used per physical channel.

The mean message latency is composed of the mean network latency, $\bar{t}_r$, and the mean waiting time, $\overline{w}_s$, in the source node. However, to capture the effects of virtual channels multiplexing, the mean message latency has to be scaled by a factor, $\bar{l}$, representing the average degree of virtual channels multiplexing. Therefore, we can write [3]

$$Latency = (\bar{t}_r + \overline{w}_s)\bar{l} \qquad (1)$$

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 900-903, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The average number of channels that a message crosses along each of the n dimensions and across the network, $\bar{k}$ and $\bar{d}$ respectively, are given by [3]

$$\bar{k} = (k-1)/2 \tag{2}$$

$$\bar{d} = n\bar{k} \tag{3}$$

Consider a message that has was transmitted successfully through the network. Since a message crosses, on average, $\bar{d}$ channels, the distribution of the network latency in the case of a successful transmission can be written as

$$T_S(x) = \text{Prob}(m + \bar{d} + d \cdot w \le x) \tag{4}$$

where w denoting the waiting time at a channel. Since the waiting times at two successive channels are independent of each other and since the Laplace-Stieltjes Transform (LST) of the sum of two independent random variables is equal to the product of their LSTs [1], the LST of $T_S(x)$ is given by

$$T_S^*(s) = \int_0^\infty e^{-sx} dT_S(x) = M^*(s) e^{-s\bar{d}} W^*(s)^d \tag{5}$$

Consider a message that experiences a timeout at the *i*-th hop channel. The LST of the network latency in the case of a transmission failure due to a timeout at the *i*-th hop channel can be written as

$$T_{F_i}^*(s) = e^{-s(2(i-1)+\tau)} W^*(s)^{(i-1)} \tag{6}$$

Let ψ_i be the number of channels that a message can select at its *i*-th hop, and, P_t be the probability that message experiences a timeout at channel. If P_l denotes the probability that *l* virtual channels at a physical channel are busy, the probability, P_{F_i}, that a message suffers a timeout at the *i*-th hop channel can be expressed as

$$P_{F_i} = P_t^{\psi_i} P_L^{\psi_i} \prod_{j=1..i-1} (1 - P_t^{\psi_j} P_L^{\psi_j}) \tag{7}$$

Given that a transmission failure can be caused by a timeout at any of the $\bar{d}$ channels along the message path, the LST of the network latency in case of a transmission failure is given by

$$T_F^*(s) = \sum_{i=1..\bar{d}} P_{F_i} T_{F_i}^*(s) \tag{8}$$

The probability of a successful transmission, P_S, and that of transmission failure, P_F, are simply given by

$$P_S = \prod_{i=1..\bar{d}} (1 - P_t^{\psi_i} P_L^{\psi_i}) \tag{9}$$

$$P_F = 1 - P_S = 1 - \prod_{i=1..\bar{d}} (1 - P_t^{\psi_i} P_L^{\psi_i}) \tag{10}$$

The network latency of a single transmission try, taking into the network latency in case of a transmission success and transmission failure, can be written as

$$T^*(s) = P_S T_S^*(s) + \sum_{i=1..\bar{d}} P_{F_i} T_{F_i}^*(s) \tag{11}$$

Let t_r be a random variable denoting the network latency seen by a message after r re-transmission attempts. Sine a new re-transmission is delayed for a time gap of v cycles and it is independent of the previous re-transmissions, we can write

$$T_r^*(s) = \sum_{r=1..\infty} P_S P_F^{r-1} T^*(s)^r V^*(s)^{r-1} = P_S T^*(s)/(1 - P_F T^*(s) V^*(s)) \quad (12)$$

The number of channels that a message can use at its i-th hop is found to be [3]

$$\psi_i = \sum_{l=0..j} l P_i^j \qquad j\bar{k} + 1 \le i \le (j+1)\bar{k} \quad (0 \le j \le n-1) \qquad (13)$$

$$P_i^j = \binom{n}{j} N_0^{\bar{k}-1}(i-1-j\bar{k},\ n\text{-}j) \Big/ \sum_{l=0..j} \binom{n}{l} N_0^{\bar{k}-1}(i-1-l\bar{k},\ n-l) \quad (14)$$

$$N_p^{p+q-1}(r,m) = \sum_{l=0..m} (-1)^l \binom{m}{l} \binom{r-mp-lq+m-1}{m-1} \qquad (15)$$

Modelling a channel as an M/M/1 queue with impatient customers and with deterministic impatient time yields a simple and practical model that exhibits a reasonable degree of accuracy. The rate of traffic on each channel is given by

$$\lambda_c = \sum_{i=0..\bar{d}-1} \prod_{j=1..i} (1 - P_t^{\psi_j} P_L^{\psi_j}) \lambda_0 \text{ , where } \lambda_o = \lambda_g / nP_S \quad (16)$$

Following the suggestions proposed of [4], the mean waiting time and probability of timeout at a given channel can be approximated as

$$\overline{w} = \frac{\dfrac{\lambda_c \bar{t}^2}{(1-\lambda_c \bar{t})} - \left(\dfrac{\bar{t}}{1-\lambda_c \bar{t}} + \lambda_c \bar{t} \tau \right) e^{\frac{-(1-\lambda_c \bar{t})\tau}{\bar{t}}} - P_t \tau}{\left(1 - \lambda_c^2 \bar{t}^2 e^{\frac{-(1-\lambda_c \bar{t})\tau}{\bar{t}}} \right)(1 - P_t)} \qquad (17)$$

$$P_t = [(1-\lambda_c \bar{t}) \lambda_c \bar{t} e^{\frac{-(1-\lambda_c \bar{t})\tau}{\bar{t}}}] / [1 - \lambda_c^2 \bar{t}^2 e^{\frac{-(1-\lambda_c \bar{t})\tau}{\bar{t}}}] \qquad (18)$$

Modelling the local queue in the source as an M/M/1 queue with a mean arrival rate λ_g / L and a mean service time $\bar{t}_r$ yields the mean waiting time as [1]

$$\overline{w}_s = (\lambda_g / L)\bar{t}_r^2 / (1 - (\lambda_g / L)\bar{t}_r) \qquad (19)$$

The probability, P_l, that l virtual channels at a given physical channel are busy is determined using a Markovian model (see [3] for a more detailed discussion). In the steady state, the model yields the following probabilities.

$$q_l = \begin{cases} 1 & l = 0 \\ q_{l-1} \lambda_c \bar{t} & 0 < l < L \\ q_{l-1} \lambda_c / (1/\bar{t} - \lambda_c) & l = L \end{cases} \qquad (20)$$

$$P_l = \begin{cases} 1/\sum_{l=0..L} q_l & l = 0 \\ P_{l-1}\lambda_c \bar{t} & 0 < l < L \\ P_{l-1}\lambda_c /(1/\bar{t} - \lambda_c) & l = L \end{cases} \quad (21)$$

The average degree of virtual channels multiplexing is given by [3]

$$\bar{l} = \sum_{i=1..L} i^2 P_i / \sum_{i=1..L} iP_i \quad (22)$$

There are inter-dependencies between the different variables of the model, and these are solved iteratively [1]. Fig. 1 depicts latency results predicted by the model and simulation in the 2-D torus (with N=64 nodes; M=32 and 64 flits; L=1 and 2 virtual channels; τ =32 and 64 cycles; V=32 and 64 cycles). The model predicts latency with a good accuracy under light and moderate traffic. However, its accuracy degrades near the saturation point due to the approximations that were used to develop the model, e.g. assumption (ii).

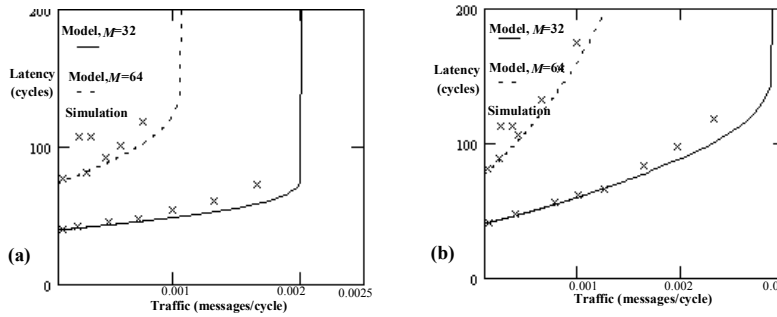


Fig. 1: Latency from the model and simulation in the 2-D torus. a) L=1, b) L=2.

3 Conclusion

This paper has presented a new analytical model for fully-adaptive routing with deadlock-recovery, based on the Compressionless routing framework [2], to predict message latency in wormhole-routed k-ary n-cubes. Simulation experiments have revealed that the analytical model predicts latency with a good degree of accuracy.

References

[1] L. Kleinrock, *Queueing Systems: Theory*, vol. 1, J. Wiley, New York, 1975.

[2] J. Kim, A. Chien and Z. Liu, Compressionless routing: A framework for adaptive and fault-tolerant routing, *IEEE TPDS* 8(3), 1997, pp. 229- 244.

[3] M. Ould-Khaoua, An analytical model of Duato's fully-adaptive routing in k-ary n-cubes, *IEEE Trans. Computers*, 44(12), 1999, pp 1-8.

[4] H.C. Tijms, Stochastic modelling and analysis: A computational approach, J. Wiley, 1986.

Analysis of Pipelined Circuit Switching in Cube Networks

Geyong Min and Mohamed Ould-Khaoua

Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, U.K.
Email: {geyong , mohamed}@cs.strath.ac.uk

Abstract. This paper proposes the first analytical model of pipelined circuit switching (PCS) in cube networks. The model uses a Markov chain to analyse the backtracking actions of the header flit during the path set-up phase in PCS. One of the main features of the present model is its ability to capture the effects of using virtual channels. The validity of the model is demonstrated by comparing analytical results to those obtained through simulation experiments.

1 Introduction

Several recent studies have revealed that pipelined circuit switching (PCS) can provide superior performance characteristics over wormhole switching because it combines the advantages of both circuit switching and wormhole switching [2], [4]. In PCS, a reserved path from the source to the destination is set up prior to the transmission of the data as in circuit switching. However, PCS differs in the way that paths are established. When the header cannot progress because all the required *virtual channels* are busy, it releases the last reserved virtual channel by backtracking to the preceding node, then continues its search from the node to find an alternative path to the destination. Since seized channels are released when blocking occurs, deadlock cannot emerge during message routing in PCS. Thus, unlike in wormhole switching, fully adaptive routing can be cheaply implemented in PCS.

This paper presents the first analytical model of PCS in hypercubes (or cubes for short). The model uses a Markov chain to calculate the mean time to set up a path, and M/G/1 queueing systems to compute the mean waiting time that a message experiences at a source before entering the network. Results from simulation show close agreement with those predicted by the model.

2 Analysis

The model is based on the following assumptions: (1) Message destinations are uniformly distributed across the network nodes. (2) Nodes generate traffic independently of each other, which follows a Poisson process with a mean rate of λ_g messages/cycle. (3) Message length is M flits, each of which requires one cycle to cross from one router to the next. (4) The local queue in the source node has infinite capacity. Moreover, messages at the destination node are transferred to the local *processing element* as soon as they arrive at their destinations. (5) Each physical channel is divided into V $(V \geq 1)$ virtual channels. (6) Exhaustive Profitable Backtracking (EPB) [4] routing protocol is used.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 904-908, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The mean message latency is composed of the mean network latency, S, that is the time to cross the network, and the mean waiting time seen by the message in the source node, W_s. However, to model the effects of virtual channels time-multiplexing, the mean message latency has to be scaled by a factor, $\overline{V}$, representing the average degree of virtual channels multiplexing at a given physical channel. Therefore, we can write [6]

$$Latency = (S + W_s)\overline{V} \tag{1}$$

Under the uniform traffic pattern, the message whose destination is i $(1 \le i \le n)$ hops away, can reach $\binom{n}{i}$ nodes out of a total of $(N-1)$ nodes in the network. The average number of channels, d, that a message visits to reach its destination is therefore given by

$$d = \frac{\sum_{i=1}^{n} i \binom{n}{i}}{N-1} = \frac{n}{2}\frac{N}{N-1} \tag{2}$$

The mean network latency, S, consists of two parts: the mean time to set up a path, C, and the actual message transmission time. Thus, S can be written as

$$S = C + d + M \tag{3}$$

In order to calculate the mean path set-up time, C, we use a Markov chain [3] to model the header actions to establish the path. State π_i $(0 \le i \le d)$ in the Markov chain corresponds to the case where the header is at the intermediate node that is i hops away from the source node. Let C_i denote the expected duration to reach state π_d starting from state π_i. A transition out of state π_i to π_{i-1} implies that the header has encountered blocking and has to backtrack to the preceding node. The residual duration becomes C_{i-1}. The transition rate is the probability, P_{b_i}, that the header is blocked in the node corresponding to state π_i. However, the transition out of state π_i to π_{i+1} denotes that the header succeeds in reserving the required virtual channel and advances one hop closer to its destination. The remaining duration is C_{i+1}. The transition rate is the probability $1 - P_{b_i}$. Given that the header requires one cycle to move from one node to the next, the above argument reveals that the expected duration C_i satisfies the difference equations

$$C_i = (1 - P_{b_i})C_{i+1} + P_{b_i}C_{i-1} + 1 \qquad (1 \le i \le d\text{-}1) \tag{4}$$

where the states π_0 and π_d satisfy the following boundary conditions

$$C_0 = (1 - P_{b_0})(C_1 + 1) + P_{b_0}C_0 \tag{5}$$

$$C_d = 0 \tag{6}$$

Solving the above equations (4~6) yields the expected duration time, C_0, to reach state π_d starting from state π_0. The mean path set-up time can be written as

$$C = C_0 + d \tag{7}$$

where the term d accounts for the d cycles that are required to send the acknowledgement flit back to the source.

On average, C channels are visited before a path is set up. Half of the visits occur to reserve the virtual channels in the direction leading to the destination node and another half take place in the opposite direction using the reserved channels. Since a router has n output channels and the local node generates λ_g messages per cycle, the mean arrival rate on a channel, λ_c, can be approximated

$$\lambda_c = \frac{\lambda_g C}{2n} \tag{8}$$

The probability of blocking, P_{b_i}, depends on the header's current network position. A header is blocked at the intermediate node that is i hops away from the source if all possible virtual channels at the remaining $(d-i)$ dimensions are busy. Let P_V denote the probability that V virtual channels at a given physical channel are busy (P_V is determined below). The probability, P_{b_i}, can be written as

$$P_{b_i} = (P_V)^{d-i} \qquad (0 \le i \le d-1) \tag{9}$$

The probability, P_t $(0 \le t \le V)$, that t virtual channels at a given physical channel are busy, can be determined using a Markovian model [1]. In the steady state, the model yields the following probabilities.

$$q_t = \begin{cases} 1 & t = 0 \\ q_{t-1}\lambda_c S & 0 < t < V \\ q_{t-1}\lambda_c / (1/S - \lambda_c) & t = V \end{cases} \tag{10}$$

$$P_t = \begin{cases} 1/\sum_{l=0}^{V} q_l & t = 0 \\ P_{t-1}\lambda_c S & 0 < t < V \\ P_{t-1}\lambda_c / (1/S - \lambda_c) & t = V \end{cases} \tag{11}$$

The average degree of multiplexing of virtual channels, that takes place at a given physical channel, is given by [1]

$$\overline{V} = \frac{\sum_{t=0}^{V} t^2 P_t}{\sum_{t=0}^{V} t P_t} \tag{12}$$

To determine the mean waiting time, W_s, that a message experiences in the source node before entering the network, the injection channel is treated as an M/G/1 queue [5]. Since a message in the source node can enter the network through any of the V virtual channels, the mean arrival rate on each virtual channel is λ_g / V. Adaptive routing distributes traffic evenly across the network, and the mean service time at each channel is approximated by the mean network latency, S [6]. Since the minimum mean network latency is $M + 3d$ in the absence of blocking, the variance of the service time distribution can be approximated as [6]

$$\sigma^2 = (S - M - 3d)^2 \tag{13}$$

As a result, the mean waiting time becomes

$$W_s = \frac{(\lambda_g / V) S^2 \left(1 + (S - M - 3d)^2 / S^2\right)}{2\left(1 - (\lambda_g / V) S\right)} \tag{14}$$

3 Model Validation

The above model has been validated by means of a discrete-event simulator. Fig. 1 depicts mean message latency results predicted by the above model plotted against those provided by the simulator in the 64 and 256 nodes cube networks. The figure reveals that the simulation results closely match those predicted by the analytical model in the steady state regions.

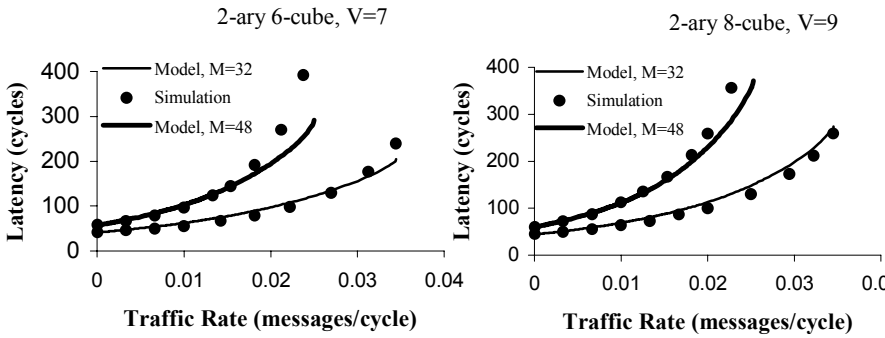


Fig. 1: Latency predicted by the model and simulation, *n*=6 and 8, *V*=7 and 9.

4 Conclusion

This paper has presented an analytical model of PCS in cube networks augmented with virtual channels. The simplicity of the model makes it a practical and cost-effective evaluation tool. The next step in our work is to develop a model for a high-radix *k*-ary *n*-cube.

References

1. Dally, W.J.: Virtual channel flow control. IEEE Trans. Parallel & Distributed System. 2 (1992), 194-205
2. Duato, J., Yalamanchili, S., Ni, L.: Interconnection networks: An engineering approach. IEEE Computer Society Press (1997)
3. Feller, W.: An introduction to probability theory and its applications, Vol. 1, John Wiley, New York, (1967)

4. Gaughan, P.T., Yalamanchili, S.: A family of fault-tolerant routing protocols for direct multiprocessor networks. IEEE Trans. Parallel & Distributed System. 5 (1995), 482-497
5. Kleinrock, L.: Queueing Systems Vol. 1, John Wiley, New York (1975)
6. Ould-Khaoua, M.: A performance model for Duato's fully adaptive routing algorithm in k-ary n-cubes. IEEE Trans. Computers. 12 (1999), 1-8

A New Reliability Model for Interconnection Networks[1]

Vicente Chirivella, Rosa Alcover

Department of Statistics and Operation Research,
Polytechnic University of Valencia,
Camino de Vera s/n, 46020 Valencia, Spain
{vchirive, ralcover}@eio.upv.es

Abstract. The traditional approach to study fault-tolerance in multicomputer interconnection networks consists of determining the worst possible combination of faulty components that causes a network failure, and then assuming that this will occur. But the worst possible combination does not always occur, and the routing algorithm allows the network to work in the presence of a greater number of failures. Thus network reliability parameters computed according to the traditional approach will be underestimated. In this paper we propose a new methodology to compute accurately the reliability function. The reliability parameters have been computed for an interconnection network with mesh topology, taking into account size, routing algorithm, failure and repair rates of the network channels and coverage.

1 Introduction

Nowadays the growing necessity of computing power has led to computer engineers to design multicomputers with a large number of processing units. As the number of components in a multicomputer increases, the probability of one or more component faults also increases. Therefore, along with performance, dependability characterization of multicomputers is essential to evaluate their effectiveness for commercial, scientific and critical-mission applications. Dependability is a generic term used to address reliability, availability, security, maintainability and other related issues [1]. The interconnection network, the subsystem that supports the message-passing mechanism, becomes a key issue in determining such dependability.

With the purpose of being able to guarantee system dependability, the objective is to design an interconnection network that works in the presence of faulty components. The designs of fault-tolerant interconnection networks can be divided into two categories: dynamic and static. Dynamic designs have redundant components and switches that allow the reconfiguration of the network and the preservation of the original topology [2]. Obviously, this solution leads to an excessively high cost as the number of nodes increases. The static design does not require the use of additional network components. The static approach takes advantage of the alternative paths existing in the network by using fault-tolerant routing algorithms [3], [4]. These algorithms bypass faulty components in the network. However, the maximum number of faults supported by these algorithms is bounded [3]. Static design has been

[1] This work was supported by the Spanish CICYT under Grant TIC97-0897.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 909-917, 2000.
© Springer-Verlag Berlin Heidelberg 2000

traditionally preferred by designers. Thus, we have evaluated the interconnection network reliability when using a static design.

Many authors have designed fault-tolerant routing algorithms. These studies obtain the worst possible combination of failures that the routing algorithm can support, and then they assume that it will occur [4]. This is the traditional approach. However, the worst possible combination does not always occur, and the routing algorithm is usually able to route in the presence of a larger number of failures.

In this paper we measure the differences between approaches, the traditional one and a more accurate one. With this objective, in section 2 we propose a new methodology for reliability prediction of interconnection networks based on a very useful statistical tool: the continuous-parameter Markov chains [5]. This methodology takes into account topology, network size and routing algorithm used, and allows us to measure the effect of the routing algorithm on the network reliability. Then, in section 3 we apply our methodology to a network with 2D mesh topology. In this section we also propose a model to compute the reliability function and the mean time to network failure, and compare the obtained results with both approaches. Finally, in section 4 some conclusions are drawn. We will show that network reliability parameters obtained with the traditional approach are always underestimated.

2 A Methodology to Evaluate Reliability Based on Markov Chains

The proposed methodology is based on Markov chains. They provide very flexible, powerful, and efficient means for the description and analysis of dynamic system properties.

The necessary tasks to apply the reliability methodology in the field of interconnection networks can be summarized in the following steps:

1- Define the Interconnection Network Fault Model. This step requires the interconnection network selection and the hypothesis assumed establishment on its operation. As it is well-known, a network is defined when its topology, flow control mechanism and routing algorithm are specified. On the other hand we must establish the hypothesis assumed in the network operation when a failure occurs.

2 - Select the network dependability parameters. In this step, we must select the dependability parameters that will quantify the reliability characteristic that we want to study. There is a large group of statistical parameters to evaluate reliability [6]. Some of these parameters are adequate to measure the reliability characteristics of gracefully degraded systems. Gracefully degrading systems react to a detected failure by reconfiguring to a state that may have a decreased level of performance.

3 - Define the network states and the transition rates. Now, we must specify the Markov chains used for model the network functioning. For this, it is necessary to define the states that represent the network operation and to establish the transitions between them. The network states are defined taking into account the reliability parameters and according to the number of faulty channels in the network.

4 - Compute the values of the network reliability parameters. In this step we must solve the system of differential equations that govern the Markov chain [5]. It provides the state probabilities. Thus, the expressions of the network reliability parameters selected can be obtained analytically.

5 - Analyze the results. Finally the results must be analyzed.

3 Applying the Reliability Methodology

In this section we apply the analytical methodology summarized in section 2. First, in section 3.1 we propose the network fault model (step1). Then, in section 3.2 we propose two models to compute the reliability function and the mean time to failure for an interconnection network with mesh topology. One model is proposed from the traditional point of view and the other under our approach (steps 2 to 4). Finally in section 3.3 we obtain and analyze the results (step5).

3.1 Fault Model

Step1. Wormhole switching is the prevalent technique in the current generation of message passing multicomputers [7]. When wormhole is used, low-dimensional meshes and k-ary n-cubes achieve a higher performance than other topologies as hypercubes [8]. Many recent experimental and commercial multicomputers, as Intel Paragon and MIT Reliable Router [9], use a mesh. A mesh is a two dimensional grid with k-nodes in each dimension. In this paper we study the reliability of a network with 16x16-mesh topology, and wormhole as flow control mechanism. Concerning to the routing algorithm, a good fault-tolerant routing algorithm should be simple, use few virtual channels, support maximum adaptivity in routing and use minimal paths when possible. Other desirable features are deadlock-freedom, good performance under no-fault scenarios, and the ability to handle a large number of faulty components [10]. Duato's Double East Last West Last fault-tolerant routing algorithm [3] possesses most of these desirable characteristics, and hence, is the routing algorithm studied in this paper.

Concerning to the assumed hypotheses on network operation, we suppose that a network fails when one or more nodes cannot communicate with each other, either because there are no physical links between them, or because the routing algorithm cannot select a route to reach the destination node. We also assume that if there is a selectable path between two nodes, the path will be selected. Nodes are reliable, and only the failures of channels are studied in this paper. Some faulty channel combinations may disconnect the network. Thus, if the network is disconnected, the network fails. A failed channel simply ceases to work and the nodes at the end of a malfunctioning channel stop using that channel. Fault-tolerant routing algorithms automatically recover the system from the occurrence of some channel failures during normal operation. The recovery consists of the detection of the fault, the identification of the faulty component, the correction of the errors induced by the fault, and notification to the neighboring nodes that there is a faulty component in the network. However, fault detection mechanisms are not perfect, and they can fail with a certain probability. The probability of system recovery when a fault occurs is called *coverage, C* [11]. In this work, this probability is assumed to be constant. Finally, the network is repairable. When the interconnection network is repaired, it is completely repaired, replacing all the faulty components.

On the other hand, the modeling of the interconnection network has been based on continuous-parameter Markov chains. For this, we have considered exponential distribution with parameter λ for the channel failure times, exponential distribution

with μ parameter for the network reparation times, and a uniform distribution of message destinations. The chosen values for these parameters are the following. The mean time to channel failure ($1/\lambda$) is measured in months, its typical value being between 1 and 6 months. The lowest value corresponds to multicomputers assembled in cabinets, while the highest one corresponds to machines mounted in rooms and wired externally. The mean time to network repair ($1/\mu$) is measured in hours. We have considered 3, 6, 12, 24, 48 and 72 hours. The lowest value corresponds to military applications, while the highest value correspond to non-critical applications. Finally, the chosen values for coverage (C) are 0'95, 0'99 and 0'999. For instance, 0'95 means that the probability of failure recover when a channel fails is 0'95. These parameters are used in the following steps.

3.2 Computing Reliability Parameters

Step2. For the sake of clarity, in this paper we have selected a reduced set of reliability parameters: the reliability function and the mean time to network failure. The reliability function ($R(t)$) at time t provides the probability that the network works correctly until this time t, given that it was operational at time 0. The mean time to network failure (*MTTF*) is the mean time at which the network first fails, and can be obtained integrating the reliability function.

Step3. Now, we must specify the Markov chains used for modeling the two approaches proposed in this paper. First, we specify the traditional approach and its transition rates. Then, we will do the same with our approach, and will pay special attention to the expressions of the transition rates between states. In our paper, we define the network states taking into account the *MTTF* and according to the number of network faulty channels. The following states have been defined: *correct state*, when there is no faulty component; *degraded state*, when there is a faulty channel, but the fault detection mechanisms have detected the fault and the routing algorithm can transmit messages between any pair of nodes; and *failed state*, when the failure of the network occurs, whatever its cause is. With these states, we propose two models (Fig.1) to compute the network reliability function. The model on the left corresponds to the traditional approach, while the other one fits to our approach.

The traditional approach to fault tolerant routing on a 2D mesh assumes that the routing algorithm can support a single failure; that is, the network fails with the failure of the second channel (worst combination). This is the case when the two channels connected to a node in the mesh border fail. Thus, the transition rates among states depend on the size of the mesh but not on the algorithm used for routing. As it is considered that the algorithm cannot route after the second failure, there are no differences among the fault-tolerant routing algorithms proposed by different authors.

The model on the left, shown in Fig.1, allows us to obtain the reliability function of the interconnection network. From the *correct state C.S.*, the network changes to the *degraded state D.S.1* (transition rate $a1$) if a failure occurs and the fault is covered, or to the *failed state F.S.* (transition rate $a2$) if the fault is not covered. In the *degraded state*, the network changes to the *correct state* if repaired (μ) or to the *failed state* when the next failure occurs ($b1$). For example, in a 16x16 mesh, the transition rate $a1$ is the product of the number of channels (480), the coverage and the failure rate of a channel ($a1=480C\lambda$). The transition rate $a2$ is the product of the number of

channels, the probability of failure in the fault detection mechanism and the failure rate of a channel ($a2=480(1-C)\lambda$). The transition rate *b1* is the product of the number of remaining non-faulty channels and the failure rate of a channel ($b1=479\lambda$). This rate is not a function of coverage because the network fails with the second failure, regardless of whether the fault is covered o not.

It must me noticed that the interconnection network can work with more faulty components than the ones allowed in the worst case. This is the new approach we propose in this paper. In order to limit the model complexity while keeping a high accuracy, we assume that after the fifth failure the performance of the network is too low and the multicomputer is turned off. Therefore, we only include the degraded states from 1 to 4 these values being the number of failed channels at each moment.

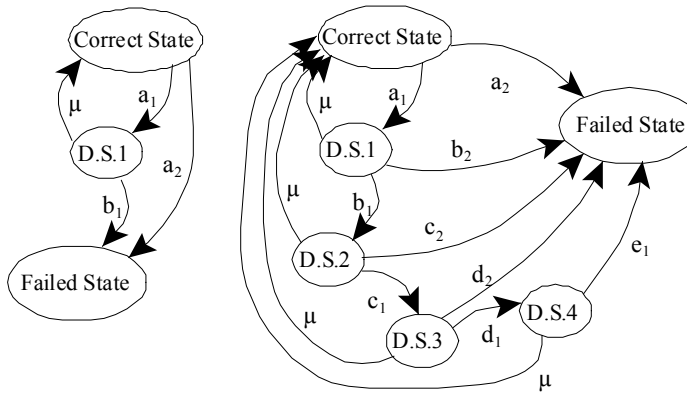


Fig. 1. Reliability model for traditional (*left*) and new approach (*right*).

The transition rates depend on the size of the mesh and on the routing algorithm used. Effectively, as the algorithm can route messages if certain combinations of failure locations occur, the transition probabilities will depend on the fault-tolerant routing algorithm chosen. The state diagram on the right in Fig.1 shows the network states and the transitions between them when our approach is used. The network starts in the correct state and evolves to the degraded state when a fault occurs if: (1) the failure is covered and, (2) the routing algorithm can still establish communication between any pair of nodes. The network changes to the failed state if at least one of the two conditions fail. The difference with the reliability model for the traditional approach is that from the *degraded state D.S.1*, the network can reach another *degraded state, D.S.2*. The transition rate between the two states depends on the ability of the routing algorithm to maintain communication after a new failure.

Once the failure combinations have been obtained, the transition rates between states can be computed. In our model, the expression of the transition rate $q_{i,i+1}$ from the degraded state i ($i\geq 0$) to the next degraded state $i+1$ is given by:

$$q_{i,i+1} = (N-i)rp_{i+1}C\lambda, \quad \textbf{(1)}$$

and the transition rate from the *degraded state i* to the *failed state* is given by

$$q_{i,i+1} = (N-i)(1-rp_{i+1}C)\lambda, \quad \textbf{(2)}$$

with

$$rp_{i+1} = nfc_{i+1}/(ncf_i(N-i))$$

where C is coverage; λ is the channel failure rate; N is the number of channels; rp_{i+1} is the probability of network working when a new failure takes place, knowing that i failures have already taken place in an operational network; and nfc_{i+1} is the number of combinations of $i+1$ faulty channels that do not cause the network failure, knowing that such combinations are obtained from the combinations of i faulty channels that did not cause the network failure, with the positions of a new faulty channel.

The transition rates for each state diagram in Fig.1 are shown in Table 1. They are computed according both traditional approach and our approach, for a 16x16 mesh and the fault tolerant routing algorithm DELWL.

Table 1. Transition rates for the traditional and the new approach.

Traditional Approach		
a1=480 C 8	b1=479 8	
a2=480 (1-C) 8		
New approach		
a1=480 C 8	c1=477'098 C 8	e1=476 8
a2=480 (1-C) 8	c2=(478-477'098 C) 8	
b1=478'625C 8	d1=475'9433 C 8	
b2=(479-478'625C) 8	d2=(477-475'9433C) 8	

Step4. The network reliability function, *R(t),* is the sum of the probabilities of being in one operational state at each instant of time [6], and the mean time to failure can be obtained integrating this function. The reliability function for the traditional approach is the sum of the probabilities of being in the *correct state*, *P(C.S.),* or in the *degraded state, P(D.S.1)*

$$R(t) = P(C.S.) + P(D.S.\ 1)$$

while in our approach the reliability function is the sum of the probabilities of being in the degraded states, numbered from 1 to 4, or in the correct state:

$$R(t) = P(C.S.) + P(D.S.1) + P(D.S.2) + P(D.S.3) + P(D.S.4)$$

With the values of λ, μ, and C, the reliability function and the *MTTF* has been computed and the results are used to compute the ratios shown in Fig.2.

3.3 Results

Step5. To compare the mean time to network failure, we use the ratio of the mean time to failure obtained with our approach and the mean time to failure obtained with the traditional approach. The results appear in Fig.2, represented as a function of channel failure rate, network repair rate, and coverage. As shown in the plots, the MTTF ratio is always larger than one. Therefore, the MTTF obtained with our approach is always larger than the MTTF obtained with the traditional approach. The ratio is in the range 1'4 to 49, that is, the network MTTF obtained with our approach can be up to 49 times larger than the value obtained with the traditional approach. As our reliability model is much more accurate than the traditional one, the MTTF obtained with the traditional approach is largely underestimated.

The differences between both approaches increase with failure coverage. This fact can be observed in the sequence of plots in Fig.2. This occurs because the network tends to transit to a degraded state instead of to the failed state, and then there are more opportunities to network repair. The difference between both approaches also increases when the channel failure rate diminishes, and the repair rate and coverage increases. The effects of coverage and network repair diminish as the failure rate increases. This occurs because the network state evolves quickly to the failed state, and the importance of the possibility of network repair diminishes. Only when coverage is high and the repair time is low, the degraded states are important and they mark the difference between both approaches.

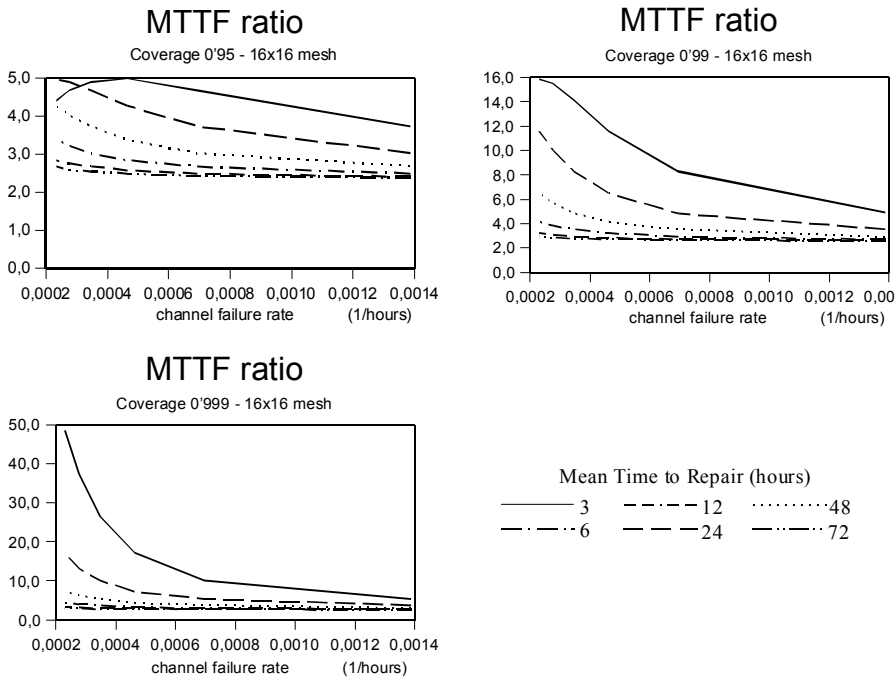


Fig. 2. MTTF ratio for a 16x16 mesh, as a function of channel failure and repair rates and coverage

We can see in the first plot of Fig.2 that the *MTTF ratio* presents a maximum. This is due to the effects of the degraded states on the network reliability. Those effects are due to the new operational states, which allow more opportunities to network repair. The importance of those effects can be reinforced or diluted by particular values of the failure and repair rates, and this is the cause for the ratio evolution.

Finally, a more detailed study of the mean time to failure, including the network size, and another dependability parameter, the steady-state availability, is available as a technical report at the GAP web server [12].

4 Conclusions

In this paper we have proposed a methodology for computing dependability in interconnection networks based on Markov chains. The Markov chains have allowed us to model the effects of the routing algorithm and the failure locations in the network. State transitions in Markov chain has been determined by considering the topology (a 16x16 mesh), the routing algorithm (DELWL) and the number and locations of faulty channels. This study has take into account that networks may work when the number of faulty channels is larger than the number of faulty channels supported by the routing algorithm.

We have determined that network reliability parameters computed using the traditional method are too conservative. It must be emphasized that the mean time to failure is always larger if it is computed according to our approach instead of the traditional approach. The differences between mean times to failure are up to 49 times larger, according to the size of the mesh. The differences grow as the coverage increases, the mean time to repair is short and mean time to failure is long. We have shown that the coverage of failures is the most important parameter on determining the mean time to network failure.

Finally, our model is close to reality and consequently provides more realistic values of network reliability parameters.

References

1. Bolch, G., Greiner, S., de Meer H. and Trivedi, K.S.: Queueing Networks and Markov Chains, Wiley-Interscience, (1998).
2. Tsai, J. and Kuo, S.: Constructions of Link-Fault-Tolerant q-ary n–cube Networks, Electronics Letters (1997), vol.33, no.12, 1025-1026.
3. Duato, J.: A Theory of Fault Tolerant Routing in Wormhole Networks, IEEE T. on Par. and Distr. Systems (1997), vol.8, no.8, 790-802.
4. Gaughan P.T. and Yalamanchili S.: A Family of Fault-Tolerant Routing Protocols for Direct Multiprocessor Networks, IEEE T. on Par. and Distr. Systems (1995), vol.6, no.5, 482-497.
5. Trivedi K.S. Probability and Statistics with Reliability, Queuing, and Computer Science Applications, Prentice-Hall (1992).
6. Beaudry M.D.: Performance-Related Reliability Measures for Computing Systems, IEEE T. on Computers (1978), vol.27, no.6, 540-547.
7. Ni, L. and McKinley, P.: A Survey of Wormhole Routing Techniques in Direct Networks, Computer (1993), vol. 26, no. 2, 62-67.

8. A. Agarwal, "Limits on interconnection network performance", IEEE T. on Par. and Distr. Systems (1991), vol. 2, no. 4, pp. 398-412
9. Dally, W.J., Dennison, L.R., Harris, D., Kan, K. and Xanthopoulus T.: The Reliable Router: A Reliable and High-performance Communication Substrate for Parallel Computers, Proc. of the Workshop on Parallel Computer Routing and Communication (1994), 241-255.
10. Vaidya, A.S., Das, R.C. and Sivasubramaniam A.: A Testbed for the Evaluation of Fault-Tolerant Routing in Multiprocessor Interconnection Networks, IEEE T. on Par. and Distr. Systems (1999), vol.10, no.10, 1052-1081.
11. Dugan, J.B. and Trivedi, K.S.: Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems, IEEE T. on Computers (1989), vol.38, no.6, 775-787.
12. Chirivella V. and Alcover R.: Improving the accuracy of reliability models for interconnection networks, Technical report (1999), http://www.gap.upv.es/index_eng.html.

A Bandwidth Latency Tradeoff for Broadcast and Reduction*

Peter Sanders and Jop F. Sibeyn

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany.
{sanders,jopsi}@mpi-sb.mpg.de.
http://www.mpi-sb.mpg.de/{~sanders,~jopsi}

Abstract. The "fractional tree" algorithm for broadcasting and reduction is introduced. Its communication pattern interpolates between two well known patterns — sequential pipeline and pipelined binary tree. The speedup over the best of these simple methods can approach two for large systems and messages of intermediate size. For networks which are not very densely connected the new algorithm seems to be the best known method for the important case that each processor has only a single (possibly bidirectional) channel into the communication network.

1 Introduction

Consider P processing units, *PUs*, of a parallel machine. *Broadcasting*, the operation in which one processor has to send a message M to all other PUs, is a crucial building block for many parallel algorithms. Since it can be implemented once and for all in communication libraries such as MPI [9], it makes sense to invest into algorithms which are close to optimal for all P and all message lengths k. Since broadcasting is sometimes a bottleneck operation, even constant factors should be considered. In addition, by reversing the direction of communication, broadcasting algorithms can usually be turned into reduction algorithms. *Reduction* is the task to compute a generalized sum $\bigoplus_{i<P} M_i$, where initially message M_i is stored on PU i and where "$\oplus$" can be any associative operator. Broadcasting and reduction are among the most important communication primitives. For example, some of the best algorithms for matrix multiplication or dense matrix-vector multiplication have these two functions as their sole communication routines [5].

We study broadcasting long[1] messages for a simple synchronous, symmetric communication model which is intended as a least common denominator of practical protocols able to support high bandwidth for long messages: It takes

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[1] For very short messages, different algorithms based on trees with large degree near the root are better, also a synchronous communication model is less attractive.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 918–926, 2000.
© Springer-Verlag Berlin Heidelberg 2000

time $t + k$ to transfer a message of size k regardless which PUs are involved. This is realistic on many modern machines where network latencies are small compared to the start-up overhead t. Both sender and receiver have to cooperate in transmitting a message. We are considering two variants. Our default is the *duplex* model where a PU can concurrently send a message to one partner and receive a message from a possibly different partner. We use the name send|recv to denote this parallel operation in pseudo-code. The more restrictive *simplex* model permits only one communication direction per processor. The broadcasting time for simplex is at most twice that for duplex communication for half as many PUs.[2] We note the cases where we can do better.

We begin our description in Sec. 2 by reviewing simple results on *pipelined* broadcasting algorithms. By arranging the PUs in a simple chain, execution time

$$T_\infty^* = k\left(1 + \mathcal{O}\left(\sqrt{tP/k}\right)\right) + \mathcal{O}(tP) \tag{1}$$

can be achieved. Except for very long messages, a better approach is to arrange the PUs into a binary tree. This approach achieves broadcasting time[3]

$$T_1^* = k\left(2 + \mathcal{O}(\sqrt{t\log(P)/k})\right) + \mathcal{O}(t\log P) \tag{2}$$

(replace "2" by "3" for the simplex model). We also give lower bounds.

The main contribution of this paper is the *fractional tree* algorithm described in Sec. 3. It is a generalization of the two above algorithms and achieves an execution time of

$$T_*^* = k\left(1 + \mathcal{O}\left(\left(\frac{t\log P}{k}\right)^{1/3}\right)\right) + \mathcal{O}(t\log P), \tag{3}$$

i.e., it combines the advantage of the chain algorithm to have a $(1 + o(1))$ factor in the k dependent term with the advantage of the binary tree algorithm to have a logarithmic dependence on P in the t dependent term of the execution time. For large P and medium k the improvement over both simple algorithms approaches a factor two (3/2 for the simplex model).

For some powerful network topologies, somewhat better algorithms are known. For Hypercubes, there is an elegant and fast algorithm which runs in time $T_{\mathrm{HC}}^* = k(1 + \sqrt{t\log(P)/k})^2 = k(1 + \mathcal{O}(\sqrt{t\log(P)/k})) + \mathcal{O}(t\log P))$ [1, 4]. However, no similarly good algorithm was known for networks with low bisection[4] bandwidth, e.g., meshes. Even for fully connected networks the best known algorithms for arbitrary P are quite complicated [2, 8]. The fractional tree algorithm does not have this problem. In Sec. 4 we explain how it can be adapted to several sparse topologies like hierarchical networks and meshes.

[2] A *couple* of simplex PUs emulate each communication of a duplex PU in two substeps. In the first substep one partner acts as a sender and the other as a receiver for communicating with other couples. In the second substep the previously received data is forwarded to the partner.

[3] Throughout this paper $\log x$ stands for $\log_2 x$.

[4] The *bisection width* of a network is the smallest number of connections one needs to cut in order to produce two disconnected components of size $\lfloor P/2 \rfloor$ and $\lceil P/2 \rceil$.

2 Basic Results on Broadcasting Long Messages

Lower Bounds. All non-source PUs must receive the k data elements, and the whole broadcasting takes at least $\log P$ steps. Thus in the duplex model there is a lower bound of

$$T_{\text{lower}} = k + t \cdot \log P \quad . \tag{4}$$

In the simplex model all non-source PUs must receive the k data elements. Hence the communication volume is at least $(P-1) \cdot k$. Even if all PUs are communicating all the time this implies a time bound of $2(1-1/P)k$. In the full paper we additionally exploit that that it takes time until PUs can start to send useful data and show a bound of

$$T_{\text{lower, simplex}} 2 \cdot (1 - 1/P) \cdot k + t \cdot (\log P - 4). \tag{5}$$

These lower bounds hold in full generality. For a large and natural class of algorithms, we can prove a stronger bound though. Consider algorithms that divide the total data set of size k in s packets of size k/s each. All PUs operate synchronously, and in every step they send or receive at most one packet. So, until step $s-1$, there is still at least one packet known only to the source. Thus, for given s, at least $s - 1 + \log P$ steps are required in the duplex model. Each step takes $k/s + t$ time. For given k, t and P, the minimum is assumed for $s = \sqrt{k \cdot t / \log P}$:

$$T^*_{\text{lower}} = k \left(1 + \sqrt{t \log(P)/k}\right)^2 \quad . \tag{6}$$

Two Simple Pipelined Algorithms. For $k \gg t$, a central idea for fast broadcasting is to chop the message into s packets of size k/s and to forward these packets in a pipelined fashion. The simplest pipelined algorithm arranges all PUs into a chain of length $P-1$. The head of the chain feeds packets downward. Interior PUs receive one packet in the first step and then in each step receive the next packet and forward the previously received packet. Fig. 1-d gives an example. It is easy to see that one gets an execution time of $T^s_\infty := (P - 2 + s) \cdot (t + \frac{k}{s})$. The optimal choice for s is $\sqrt{k(P-2)/t}$. Substituting this into T^s_∞ yields $T^*_\infty := k \left(1 + \sqrt{t \cdot (P-2)/k}\right)^2 = k \left(1 + \mathcal{O}\left(\sqrt{tP/k}\right)\right) + \mathcal{O}(tP)$.

For $k \gg tP$ the performance of this algorithm is quite close to the lower bound (4). However, since t is usually a large constant, on systems with large P we only get good performance for messages which are extremely large. We can reduce the dependence on P by arranging the PUs into a binary tree. Now every interior node forwards every packet to both successors. This needs two steps per packet. The execution time is $T^s_1 := (d + 2s) \cdot (t + \frac{k}{s})$ where d is the time step just before the last leaf receives the first packet; d is defined by the recurrence $P_i = 1 + P_{i-1} + P_{i-2}$, $P_0 = 1$, $P_1 = 2$. We have $d = \min\{i : P_i \geq P\} - 1 \approx \log_{1.62} P$. For our purposes it is sufficient to note that $d = \mathcal{O}(\log P)$. Fig. 1-a shows the tree with $P_5 = 18$ PUs. Choosing $s = \sqrt{2k \cdot d/t}$, one gets $T^*_1 :=$

$k\left(\sqrt{2}+\sqrt{d\cdot t/k}\right)^2 = k\left(2+\mathcal{O}(\sqrt{t\log(P)/k})\right)+\mathcal{O}(t\log P)$. (For the simplex model replace the two by a three.) For small and medium k this is much better than the chain algorithm, yet for large k it is almost two times slower.

3 Fractional Tree Broadcasting

The starting point for this paper was the question whether we could find a communication pattern which allows a more flexible tradeoff between the high bandwidth of a chain (i.e., a tree with degree one) and the low latency of a binary tree. We give a family of communication pattern we call *fractional trees* which have this property. Here we describe the algorithm for the duplex model in detail. As already outlined in the introduction, the duplex algorithm can be translated into a simplex algorithm running in double time. In the full paper, we explain a faster direct implementation which is able to forward a run of r packets in $2r+1$ steps and hence is only a factor $2-\frac{1}{r+1}$ times slower than on the duplex model. It turns out to be nontrivial to translate a parallel send|recv into a sequences "send, recv" or "recv, send" such that no delays or deadlocks occur.

The idea for fractional trees is to replace the node of a binary tree by a group of r PUs forming a chain. The input is fed into the head of this chain. The data is passed down the chain and on to a successor group as in the single chain algorithm. In addition, the PUs of the group cooperate in feeding the data to the head of a second successor group. Fig. 1 shows the structure of a group and several examples.

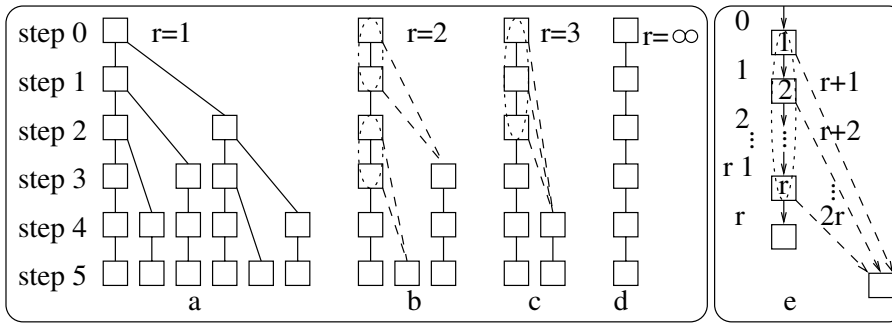


Fig. 1. Examples for fractional trees with $r \in \{1,2,3,\infty\}$ where the last PU receives its first packet after 5 steps. The case $r=1$ corresponds to plain binary trees and pipelines can be considered the case $r=\infty$. Part e) shows the communication pattern in a group of r PUs which cooperate to supply two successor nodes with all the data they have to receive. Edges are labeled with the first time step when they are active.

```
Procedure broadcastFT(r, s, 0 ≤ i < r:Integer; var D[0..s − 1]:Packet)
    recv(D[0])                                          −− wait for first packet
    pipeDown(r, 0, D)                                   −− First phase
    for k := r to s − r step r do                       −− Remaining phases
        sendRight|Recv(D[k − r + i], D[k])
        pipeDown(r, k, D)
    sendRight(D[s − r + i])

    (* send down packets D[k..k + r − 1] and receive packets D[k + 1..k + r − 1] *)
    Procedure pipeDown(r, k:Integer; var D[..]:Packet)
    for j := k to k + r − 2 do sendDown|Recv(D[j], D[j + 1])
    sendDown(D[k + r − 1])
```

Fig. 2. Pseudocode executed on each PU for fractional tree broadcasting, where i is the index of the PU within its group, s is a multiple of r, and the array D is the input on the root and the output on the other PUs. For the root PU, receiving is a no-op. For the top PU of a group receiving means receiving from any PU in the predecessor group. For the other PUs it means receiving from the predecessor in the group. Sending down means sending to the next PU in the group respectively sending to the top PU of the successor group. Sending right means sending to the top PU of the right successor group. If the successor defined by this convention does not exist, sending is a no-op.

All PUs execute the same code shown in Fig. 2. All timing considerations are naturally handled by the synchronization implicit in synchronous point-to-point communication. The input is conceptually subdivided into s/r runs of r packets each. The only nontrivial point is that the i-th member of a group is responsible for passing the i-th packet of every run of r packets to the right. The effect is that every $r+1$ steps the head of the right successor gets a run of r packets in the right order. The pause after this run is used to pass the last packet downward. Packets are passed right while the next run arrives.

As in the special case of binary trees ($r = 1$), the right successors receive data one step later than the downward successors. Therefore, optimal tree layouts are somewhat skewed. The number of nodes reachable within $d + 1$ steps is governed by the recurrence $P_i = r + P_{i-r} + P_{i-r-1}$ ($P_i = i + 1$ for $i \leq r$) so that $d = \min\{i : P_i \geq P\} - 1$. This implies $d = \mathcal{O}(r \log(P/r))$. Using this recurrence each processor can find its place in the tree in time $\mathcal{O}(d)$ and without any communication.

Performance Analysis. Having established a smooth timing of the algorithm the analysis can proceed analogously to that of the simple algorithms from the introduction. Every communication step takes time $(t+k/s)$ and $d+s\cdot(1+1/r)$ steps are needed until all s/r runs have reached the last leaf group. We get a total time of

$$T_r^s := \left(d + s\left(1 + \frac{1}{r}\right)\right)\left(t + \frac{k}{s}\right) \quad . \tag{7}$$

Using calculus one gets $s = \sqrt{kdr/(t(r+1))}$ as an optimal choice for the number of packets. Substituting this into Eq. (7) yields

$$T_r^* := k\left(1+\tfrac{1}{r}\right)\left(1+\sqrt{\tfrac{drt}{k(r+1)}}\right)^2 = k\left(1+\tfrac{1}{r}+\mathcal{O}(\sqrt{\tfrac{rt\log P}{k}})\right)+\mathcal{O}(rt\log P). \quad (8)$$

Since d depends on r in a complicated way, there seems to be no closed form formula for an optimal r. But we get a close to optimal value for r by setting $d = d' \cdot (r+1)$ and ignoring that d' depends on r.[5] We get $r \approx (k(r+1)/(d \cdot t))^{1/3}$. After rounding, these values make sense for $k(r+1) \geq d \cdot t$. For smaller k one should use $r = 1$ or even a non-pipelined algorithm. Substituting r and s into T_r^s we get a broadcasting algorithm with execution time

$$T_*^* \leq k\left(1+\left(\frac{d\cdot t}{k(r+1)}\right)^{1/3}\right)^3 = k\left(1+\mathcal{O}\left(\left(\frac{t\log P}{k}\right)^{1/3}\right)\right)+\mathcal{O}(t\log P)\ .$$

For $k \gg t \log P$ algorithm performs quite close to the lower bound (4).

Performance Examples. How does the algorithm compare to the two simple algorithms? For example, for $P = 1024$ and $k/t = 4096$ we choose $d' = d/(r+1) \approx \log P - 1 = 9$ and get $r \approx (k/(d' \cdot t))^{1/3} \approx 8$. This yields $d = 57$ and we get $s = \sqrt{4096 \cdot 57 \cdot 8/9} \approx 456$. With these values $T_r^s \approx 1.389k$. These choices are quite robust. For example, a better approximation of the optimal r yields $r = 10$ and $s = 503$ but the resulting $T_r^s \approx 1.387k$ is less than 0.2 % better.

Fig. 3 plots the achievable speedup for three different machine sizes. Even for a medium size parallel computer ($P = 64$) an improvement of up to a factor 1.29 can occur. For very large machines ($P = 16384$) the improvements reach up to factor of 1.8 and a significant improvement is observed over a large range of message lengths.

Our conclusion is that fractional tree broadcasting yields a small improvement for "everyday parallel computing" yet is a significant contribution to the difficult task of exploiting high end machines such as the ones currently build in the ASCI program. For example, Compaq plans to achieve 100TFlops with 16384 Alpha processors by the year 2004 [3].

4 Sparse Interconnection Networks

Hierarchies of Crossbars. Compaq's above mentioned 16384 PU system is expected to consist of 256 SMP modules with 64 PUs each. We view it as unlikely that it will get an interconnection network with enough bisection width to efficiently implement the hypercube algorithm. Rather, each module will only have a limited number of channels to other modules. We call such a system a 256×64 *hierarchical* crossbar. Systems with similar properties are currently build by several companies.

[5] In the program one can efficiently solve the equations numerically, e.g., using golden section search [7].

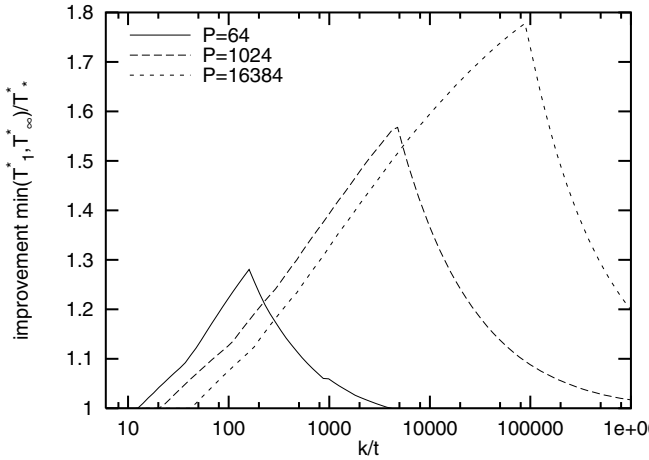


Fig. 3. Improvement of fractional tree broadcasting over the best of pipelined binary tree and sequential pipeline algorithm as a function of k/t.

We now explain how a fractional tree with group size r can be embedded into an $a \times b$ hierarchical crossbar if $b \geq r$ and if each module supports at least two incoming and outcoming channels to arbitrary other modules. A generalization to more than two levels of hierarchy is also possible.

First, one group in each module is connected to form a *global* binary tree with a nodes. Next, the $b - r$ remaining PUs in each module are connected to a form *local* fractional tree. What remains to be done is to connect the local trees by the global tree. Groups in the global tree with degree one can directly link with their local tree. Leaf groups in the global tree use one of their free links to connect to their local tree. The remaining free links are used to connect to the local trees of modules with a group in the global tree of degree two. There will be one remaining unused link which can be used to further optimize the structure.

By accepting an additional depth of $(r + 1) \log b$, we can work with one less connection per module: Use two global groups per module. The first one links to the second one and one other module. The second one links to the local tree and possibly to one other module.

Meshes. We only outline a simple case. Generalizations which are sufficient in practice should be relatively easy. A completely general treatment might turn out to be rather complicated. Assume we are given an $a \times b$ mesh and $r = r_1 \cdot r_2$ such that $a/r_1 = b/r_2$ is a power of two. We partition the mesh into submeshes of size $r_1 \times r_2$ each forming a group of the fractional tree. Now we can embed the binary tree of groups exploiting the substantial work on embedding binary trees into meshes (e.g., [10, 6]). Inside the group, the PUs are arranged in a snakelike fashion. In this way one gets an embedding with constant edge congestion. Often

it is even possible to achieve edge congestion one for bidirectional meshes. Fig. 4 gives an example where it is exploited that *H-trees* yield a complete binary tree with one leaf in every 2×2 submesh.

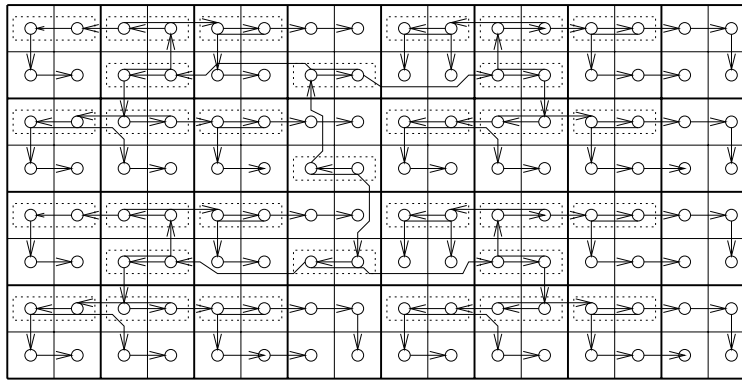


Fig. 4. Embedding of a fractional tree with $r = 2$ into an 8×16 mesh. Broadcasting on it gives edge congestion one even with x-y-routing.

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C. Ho, S. Kipnis, and M. Snir. CCL: A portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.

[2] A. Bar-Noy and S. Kipnis. Broadcasting multiple messages in simultaneous send/receive systems. In *5th IEEE Symp. Parallel, Distributed Processing*, pages 344–347, 1993.

[3] Compaq. AlphaServer SC series product brochure, 1999. `http://www.digital.com/hpc/news/news_sc_launch.html`.

[4] S. L. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[6] J. Opatrny and D. Sotteau. Embeddings of complete binary trees into grids and extended grids with total vertex-congestion 1. *Discrete Applied Mathematics*, 98:237–254, 2000.

[7] W. H. Press, S. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.

[8] Santos. Optimal and near-optimal algorithms for k-item broadcast. *JPDC: Journal of Parallel and Distributed Computing*, 57:121–139, 1999.

[9] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference*. MIT Press, 1996.

[10] P. Zienicke. Embedding of treelike graphs into 2-dimensional meshes. In *Graph Theoretic Concepts in Computer Science*, volume 484 of *LNCS*, pages 182–192. Springer, 1990.

Optimal Broadcasting in Even Tori with Dynamic Faults

Stefan Dobrev and Imrich Vrt'o*

Institute of Mathematics, Slovak Academy of Sciences
Department of Informatics, P.O.Box 56, 840 00 Bratislava, Slovak Republic
{kaifdobr,vrto}@savba.sk

Abstract. We consider a broadcasting problem in the n-dimensional k-ary even torus in the shouting communication mode, i.e. any node of a network can inform all its neighbours in one time step. In addition, during any time step a number of links of the network can be faulty. Moreover the faults are dynamic. The problem is to determine the minimum broadcasting time if at most $2n-1$ faults are allowed in any step. In [3], it was shown that the broadcasting time is at most *diameter*+$O(1)$, provided that k is limited by a polynomial of n. In our paper we drop this additional assumption and prove that the broadcasting can be always done in time *diameter +2*. The bound is the best possible.

1 Introduction

Broadcasting is the standard communication problem in interconnection networks when a node has to send a message to all other nodes. There are many applications of the broadcasting problem in parallel and distributed computing [6,8,9]. Recently, a lot of attention has been paid to fault-tolerant dissemination of information in networks [10]. In this paper we consider the shouting communication mode in which any node can inform all its neighbours in one time step. In addition, we assume that during any time step a number of links of the network can be faulty. This model was introduced by Santoro and Widmayer [11]. The problem is to determine the minimum broadcasting time if at most f faults are allowed in any time step, where f stands for the edge connectivity minus 1 of the network. For the hypercube, the problem was studied in [3,7,10] and completely solved in [5]. For the n-dimensional general torus, Chlebus, Diks and Pelc [2] proved an upper bound on the brodcasting time $O(diameter)$. De Marco and Rescigno [4] showed for the n-dimensional k-ary even torus that the broadcasting time is at most *diameter*+$O(1)$, provided that k is limited by a polynomial of n. In our paper we drop this additional assumption and prove that the broadcasting can be always done in time *diameter +2*. The bound is the best possible.

Our method previously used in [5] is related to the isoperimetric problem in graphs and can be applied to other networks.

* Supported by the VEGA grant No. 2/7007/20.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 927–930, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Model and Basic Facts

Let $C_{n,k}$ be a network of processors connected as the n-dimensional k-ary torus defined as the cartesian product of n cycles C_k, for k even. The network $C_{n,k}$ has k^n nodes, is regular of degree $2n$, with edge connectivity $2n-1$. Its diameter equals $nk/2$. The links are bidirectional. The computation is synchronous. In one time step a node is able to send its message to all its neighbours. This is called the shouting mode. In each time step at most $2n-1$ links are faulty, i.e., the message transmitted along the faulty link is not delivered. The faults are dynamic in the sense that the set of faulty links can change during the execution of the broadcast. Initially, a node of $C_{n,k}$ knows a message. This message needs to be sent to all other nodes. Our problem is to determine the minimum time to broadcast the message in the torus $C_{n,k}$, provided that the faults are distributed in the worst possible manner.

The vertices of $C_{n,k}$ are $\{-k/2, -k/2+1, ..., k/2-1\}$, and two vertices u and v are adjacent if $|u-v| \pmod k = 1$. The vertices of $C_{n,k}$ are n-tuples $(x_1, x_2, ..., x_n)$, where $-k/2 \leq x_i \leq k/2-1$, for $i = 1, 2, ..., n$. Define $\mathbf{o} = (0, 0, ..., 0)$. Let $d(u,v)$ be the distance between u and v in $C_{n,k}$. Denote $S(r) = \{v \in C_{n,k} | d(v, \mathbf{o}) = r\}$ and $V(r) = \{v \in T_{n,k} | d(v, \mathbf{o}) \leq r\} = \cup_{i=0}^{r} S(r)$. Clearly, $|S(r)|$ is the number of integer solutions of the equation $|x_1| + |x_2| + ... + |x_n| = r$, where $-k/2 \leq x_r \leq k/2-1$. Since k is even we have $|S(r)| = |S(nk/2 - r)|$, which implies $|V(r)| + |V(nk/2 - r - 1)| = k^n$. Moreover, $|S(0)| = 1, |S(1)| = 2n$ and $|S(2)| = 2n^2$, for $k \geq 6$.

Let A be a subset of vertices of $C_{n,k}$. Let $\partial(A)$ denote the set of all vertices in distance at most 1 from A in $C_{n,k}$. Bollobás and Leader [1] proved:

Lemma 1. *Let $C_{n,k}$ be the n-dimensional k-ary torus with k even, and let A be a nonempty subset of vertices of $C_{n,k}$. If $|A| = |V(r)| + \alpha|S(r+1)|$, for some r and $0 \leq \alpha < 1$, then $|\partial(A)| \geq |V(r+1)| + \alpha|S(r+2)|$.*

3 Optimal Upper Bound on the Broadcasting Time

In this Section we use Lemma 1 to bound the broadcasting time. First we state a technical lemma, proof of which will appear in the full version.

Lemma 2. *Denote*

$$X = 1 - \frac{3}{n} - \sum_{r=3}^{\lfloor \frac{nk}{4} \rfloor} \frac{4n}{|S(r)|}.$$

The value X is nonnegative for $k \geq 6$ and $n \geq k+4$ or $k \geq n \geq 10$.

Theorem 1. *Assume that $k \geq 6$ and $n \geq k+4$ or $k \geq n \geq 10$. The minimum broadcasting time T in the n-dimensional k-ary torus, for k even, with $2n-1$ dynamic link faults satisfies $T \leq nk/2 + 2$. The bound is the best possible.*

Proof. The broadcasting scheme is simple. Initially, the node **o** contains the message to be disseminated. In each time step each node sends the message to all its neighbours. The analysis follows. By A_m we will denote suitable sets of nodes which know the message after the m-th time step. Observe that the number of nodes that know the message after the m-th step is at least $|\partial(A_{m-1})|-(2n-1)$. Clearly, there exist sets A_0 and A_1 and A_2 s.t. $|A_0| = 1, |A_1| = 2$ and $|A_2| = 1 + 2n = |V(1)|$. According to Lemma 1 $|\partial(A_2)| \geq V(2)|$.

Because of the $2n-1$ faulty links, we have

$$|\partial(A_2)| - (2n-1) \geq 2n^2 + 2 = |V(1)| + \alpha_3|S(2)|,$$

where $\alpha_3 = 1 - 1/n$. Define A_3 to be a subset of nodes that know the message after the 3-rd step and satisfies $|A_3| = |V(1)| + \alpha_3|S(2)|$. Similarly, $|\partial(A_3)| \geq |V(2)| + \alpha_3|S(3)|$.

Because of the $2n-1$ faulty links, we have

$$|\partial(A_3)| - (2n-1) = |V(2)| + \alpha_3|S(3)| - (2n-1) \geq |V(2)| + \alpha_4|S(3)|,$$

where $\alpha_4 = \alpha_3 - \frac{2n}{|S(3)|}$. Define A_4 to be a subset of nodes that know the message after the 4-th step and satisfies $|A_4| = |V(2)| + \alpha_4|S(3|$.

We prove by induction that for $4 \leq m \leq \frac{nk}{2} - 1$

$$|A_m| = |V(m-2)| + \alpha_m|S(m-1)|,$$

where

$$\alpha_m = 1 - \frac{1}{n} - \frac{2n}{|S(3)|} - \frac{2n}{|S(4)|} - ... - \frac{2n}{|S(m-1)|}.$$

Assume the claim holds for some $4 \leq m-1 \leq \frac{nk}{2} - 2$, i.e.

$$|A_{m-1}| = |V(m-3)| + \alpha_{m-1}|S(m-2)|,$$

where

$$\alpha_{m-1} = 1 - \frac{1}{n} - \frac{2n}{|S(3)|} - \frac{2n}{|S(4)|} - ... - \frac{2n}{|S(m-2)|}.$$

Lemma 1 implies

$$\begin{aligned} |\partial(A_{m-1})| - (2n-1) &= |V(m-2)| + \alpha_{m-1}|S(m-1)| - (2n-1) \\ &\geq |V(m-2)| + \alpha_m|S(m-1)|, \end{aligned}$$

where $\alpha_m = \alpha_{m-1} - \frac{2n}{|S(m-1)|}$. Define A_m to be a subset of nodes that know the message after the m-th step and satisfies $|A_m| = |V(m-2)| + \alpha_m|S(m-1)|$. Then

$$\alpha_m = 1 - \frac{1}{n} - \sum_{i=3}^{m-1} \frac{2n}{|S(i)|}.$$

Now we use a dual argument. By B_m we will denote suitable subsets of nodes, which do not know the message after the m-th step. Assume that after

the $(nk/2+2)$-nd step there exists at least one node which does not know the message. Observe that the number of nodes that do not know the message after the $(m-1)$-st step is at least $|\partial(B_m)|-(2n-1)$. Clearly there exist sets $B_{\frac{nk}{2}+2}$, $B_{\frac{nk}{2}+1}$ and $B_{\frac{nk}{2}}$ such that $|B_{\frac{nk}{2}+2}|=1$, $|B_{\frac{nk}{2}+1}|=2$ and $|B_{\frac{nk}{2}}|=1+2n$. Similarly as for A_3 we determine $|B_{\frac{nk}{2}-1}|=2n^2+2$.

Now compute

$$|A_{\frac{nk}{2}-1}|+|B_{\frac{nk}{2}-1}|=|V(\frac{nk}{2}-3)|+(1-\frac{1}{n}-\sum_{r=3}^{\frac{nk}{2}-2}\frac{2n}{|S(r)|})|S(\frac{nk}{2}-2)|+2n^2+2$$

$$\geq k^n+1+2n^2(1-\frac{3}{n}-\sum_{r=3}^{\lfloor\frac{nk}{4}\rfloor}\frac{4n}{|S(r)|}).$$

By Lemma 2 the expression in the brackets is positive which implies a contradiction.

Finally, we show that there is a distribution of faults in each step which forces the brodcasting time $nk/2+2$. Consider vertices u,v such that $d(u,v)=nk/2$. Let u' (v') be a neighbor of u (v). Initially, let u' knows the message. In the first step we place faults on all edges adjacent to u', except for $u'u$. From now on we place faults on all edges adjacent to v', except for $v'v$. After $nk/2+1$ steps, the message reaches the vertex v and one additional step is necessary to complete the broadcasting. □

References

1. Bollobás, B., Leader, I., An isoperimetric inequality on the discrete torus, *SIAM J. on Discrete Mathematics* **3** (1990), 32-37.
2. Chlebus, B., Diks, K., Pelc, A., Broadcasting in synchronous networks with dynamic faults, *Networks* **27** (1996), 309-318.
3. De Marco, G., Vaccaro, U., Broadcasting in hypercubes and star graphs with dynamic faults, *Information Processing Letters* **66** (1998), 321-326.
4. De Marco, G., Rescigno, A.A., Tighter bounds on broadcasting in torus networks in presence of dynamic faults, *Parallel Processing Letters*, to appear.
5. Dobrev, S., Vrt'o, I., Optimal broadcasting in hypercubes with dynamic faults, *Information Processing Letters* **71** (1999), 81-85.
6. Fraigniaud, P., Lazard, E., Methods and problems of communication in usual networks, *Discrete Applied Mathematics* **53** (1994), 79-133.
7. Fraigniaud, P., Peyrat, C., Broadcasting in a hypercube when some calls fail, *Information Processing Letters* **39** (1991), 115-119.
8. Hedetniemi, S.M., Hedetniemi, S.T., and Liestman, A., A survey of gossiping and broadcasting in communication networks, *Networks* **18** (1986), 319-349.
9. Hromkovic, J., Klasing, R., Monien, B., Paine, R., Dissemination of information in interconnection networks (broadcasting and gossiping), in: Combinatorial Network Theory, (D.-Z. Du, D. F. Hsu, eds.), Kluwer Academic Publishers, 1995, 125-212.
10. Pelc, A., Fault tolerant broadcasting an gossiping in communication networks, *Networks* **26** (1996), 143-156.
11. Santoro, N., Widmayer, P., Distributed function evaluation in the presence of transmission faults, in: *SIGAL'90*, LNCS 450, Springer Verlag, Berlin, 1990, 358-369.

Broadcasting in All-Port Wormhole 3-D Meshes of Trees*

Petr Salinger and Pavel Tvrdík

Department of Computer Science and Engineering
Czech Technical University, Karlovo nám. 13
121 35 Prague, Czech Republic
{salinger,tvrdik}@sun.felk.cvut.cz

Abstract. In this paper, we show that 3-D meshes of trees allow an elegant and optimal one-to-all broadcast algorithm supposing routers with distance-insensitive switching and all-output-port capability.

1 Preliminaries

Let $\mathcal{B} = \{0, 1\}$ be the *binary* alphabet and let $\mathcal{B}^n = \{x_{n-1} \dots x_0 \,; x_i \in \mathcal{B}\}$, $n \geq 1$, be the set of all n-bit strings. For integer $i \geq 1$, the *i-fold concatenation* of bit α is denoted by α^i. The *empty* string is $\alpha^0 = \varepsilon$. Let $\mathcal{B}^0 = \{\varepsilon\}$. If $x \in \mathcal{B}^i$, then $|x| = i$ denotes its *length*. An undirected graph G consists of *nodes* $N(G)$ and *edges* $E(G)$. An *edge joining* 2 *adjacent* nodes u and v is denoted by $\langle u, v\rangle$. Given integer $n \geq 1$, *the complete binary tree* of *height* n, CBT_n, is defined as follows: $N(CBT_n) = \cup_{i=0}^{n} \mathcal{B}^i$ and $E(CBT_n) = \{\langle x, xa\rangle \,; |x| < n,\ a \in \mathcal{B}\}$. We assume the *standard tree node* labeling (see Figure 1(a)).

The 3-D *mesh of trees* of height n, $M3T_n$, is defined as follows:

$$N(M3T_n) = (\mathcal{B}^n \times \mathcal{B}^n \times \cup_{i=0}^{n} \mathcal{B}^i) \cup (\mathcal{B}^n \times \cup_{i=0}^{n} \mathcal{B}^i \times \mathcal{B}^n) \cup (\cup_{i=0}^{n} \mathcal{B}^i \times \mathcal{B}^n \times \mathcal{B}^n),$$

$$\begin{aligned} E(M3T_n) = &\{\langle (x, y, z), (xa, y, z)\rangle \,; |x| < n,\ |y| = |z| = n,\ a \in \mathcal{B}\} \\ &\cup \{\langle (x, y, z), (x, ya, z)\rangle \,; |y| < n,\ |x| = |z| = n,\ a \in \mathcal{B}\} \\ &\cup \{\langle (x, y, z), (x, y, za)\rangle \,; |x| = |y| = n,\ |z| < n,\ a \in \mathcal{B}\}. \end{aligned}$$

Hence, $|N(M3T_n)| = 4 \cdot 2^{3n} - 3 \cdot 2^{2n}$ and $|E(M3T_n)| = 6 \cdot 2^{3n} - 6 \cdot 2^{2n}$. Also, $M3T_n \subset CBT_n \times CBT_n \times CBT_n$. $M3T_n$ can be viewed as a 3-D mesh $2^n \times 2^n \times 2^n$ whose each node is a leaf of exactly 1 x-tree, 1 y-tree, and 1 z-tree (see Figure 1(c)). $M3T_n$ can be *canonically decomposed* into 2^{3i} 3-D submeshes of trees, denoted by $\mathrm{sub}M3T_{n-i}(x, y, z)$, where $x, y, z \in \mathcal{B}^i$. The 2-D *mesh of trees* ($M2T_n$) is defined similarly (see Figure 1(b)).

Given a connected network G and a designated node s possessing a packet, the task of a *one-to-all broadcast* (OAB) in G with the *source* s is to deliver the packet from s to every other node in G. We assume that each node has a router capable of *distance-insensitive* switching, such as wormhole or circuit switching. We denote such networks as WH networks. In each round, all paths used by

* This research has been supported by MŠMT research program #J04/98:212300014, by ČVUT grant #300009203, and by FRVŠ grant #580/2000.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 931–934, 2000.
© Springer-Verlag Berlin Heidelberg 2000

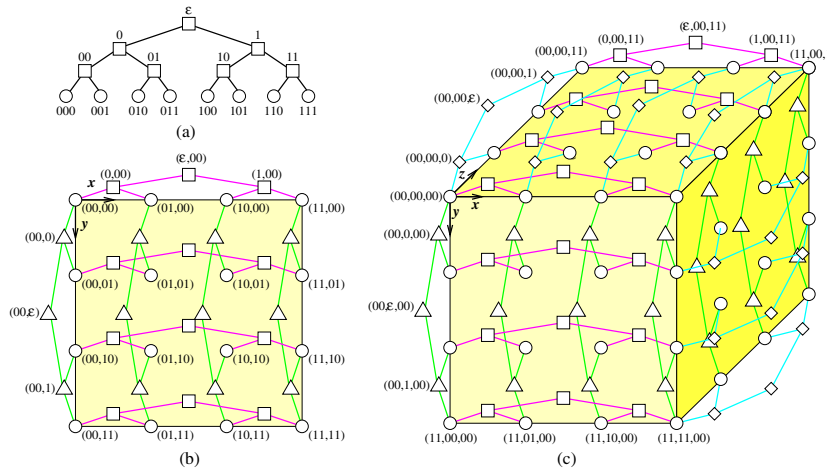


Fig. 1. Node labeling in (a) CBT_3, (b) $M2T_2$, and (c) $M3T_2$.

copies of the packet are *pairwise link-disjoint*. Links can be *half-* or *full-duplex*. We assume *all-output-port* capability, i.e., in one round a router can inject copies of the packet from the node into the network via all output links simultaneously.

2 Previous and Related Work

Several OAB algorithms have been developed for all-port WH 2-D meshes and tori, e.g., [2,5]. 2-D meshes of trees have been studied in [1,3,4]. Paper [1] gives algorithms for OAB and AAB in 1-port store-and-forward full-duplex combining model. An optimal OAB algorithm and asymptotically optimal algorithms for AAB, in 1-port WH *node-disjoint* and both half-duplex and full-duplex combining models are presented in [3]. An OAB algorithm in all-output-port WH model is described in [4]. It achieves the optimal number of rounds if the source of the OAB is a mesh node, level-1 tree node, or a root. This represents 67% of nodes. In the remaining cases, it needs 1 additional round.

3 The Main Result

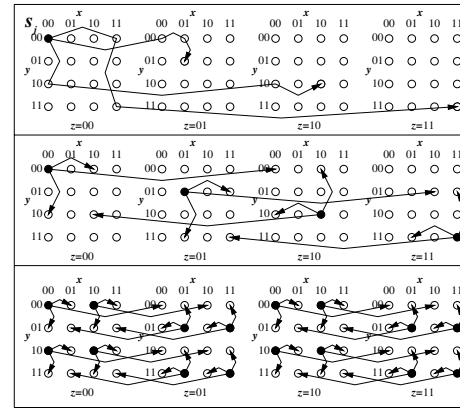
Algorithm OAB(n, s_j):

if Even(n) **then** {
  if $j \neq n$ **then** /* s_j is not a mesh node */
    s_j sends the packet to $(0^n, 0^n, 0^n)$;
  apply A(n);
  apply C(n); }
else { /* n is odd */
  apply B(n, s_j);
  for all $x, y, z \in \mathcal{B}$ **do_in_parallel**
    apply A$(n-1)$ in sub$M3T_{n-1}(x, y, z)$; /* in all 8 submeshes */
  apply C(n); };

The OAB algorithm uses the standard shortest-path routing. Without loss of generality, we assume source node $s_j = (0^j, 0^n, 0^n)$.

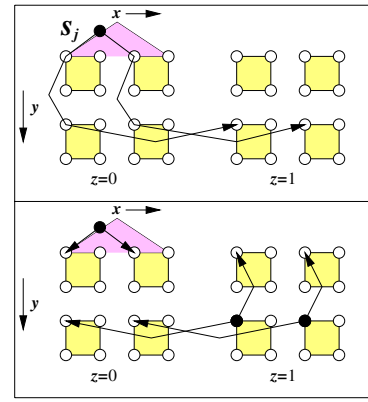
Procedure $\mathbf{A}(n)$. Assumptions: n is *even* and the source is *mesh* node $s = (0^n, 0^n, 0^n)$. In the first three rounds of procedure $\mathbf{A}(n)$, 1 mesh node in each of 64 sub$M3T_{n-2}(x, y, z)$, $x, y, z \in \mathcal{B}^2$ is informed. In further 3 rounds, each of these 64 nodes informs other 64 nodes in its 64 sub$M3T_{n-4}$'s, and so on, up to the granularity of individual mesh nodes. Hence, after completion of procedure $\mathbf{A}(n)$, all the mesh nodes are informed.

procedure $\mathbf{A}(n)$:
for $i = n-2$ **downto** 0 **by 2 do**
  for all $x, y, z \in \mathcal{B}^{n-2-i}$ **do_in_parallel** {
    $(x000^i, y000^i, z000^i)$ sends the packet
      to $(x110^i, y110^i, z110^i)$
        via $(x110^i, y, z000^i)$
    and to $(x100^i, y100^i, z100^i)$
        via $(x000^i, y100^i, z)$
    and to $(x010^i, y010^i, z010^i)$
        via $(x0, y000^i, z010^i)$;
  for all $u, v \in \mathcal{B}$ **do_in_parallel**
    $(xuv0^i, yuv0^i, zuv0^i)$ sends the packet
      to $(x\bar{u}v0^i, yuv0^i, zuv0^i)$
    and to $(xuv0^i, y\bar{u}v0^i, zuv0^i)$
    and to $(xuv0^i, yuv0^i, z\bar{u}v0^i)$;
  for all $u, v, w, t \in \mathcal{B}$ **do_in_parallel**
    $(xut0^i, yvt0^i, zwt0^i)$ sends the packet
      to $(x\bar{u}t0^i, yvt0^i, zwt0^i)$
    and to $(xut0^i, y\bar{v}t0^i, zwt0^i)$
    and to $(xut0^i, yvt0^i, zw\bar{t}0^i)$;};



Procedure $\mathbf{B}(n, s_j)$. The source node is $s_j = (0^j, 0^n, 0^n)$, $0 \le j \le n$. Procedure $\mathbf{B}(n, s_j)$ performs 2 rounds and informs 1 mesh node in each sub$M3T_{n-1}(x, y, z)$, $x, y, z \in \mathcal{B}$.

procedure $\mathbf{B}(n, s_j)$:
/* Round 1 */
s_j sends the packet
  to $(00^{n-1}, 10^{n-1}, 10^{n-1})$
  and to $(10^{n-1}, 10^{n-1}, 10^{n-1})$;
/* Round 2 */
parbegin
  s_j sends the packet
    to $(00^{n-1}, 00^{n-1}, 00^{n-1})$
    and to $(10^{n-1}, 00^{n-1}, 00^{n-1})$;
  $(00^{n-1}, 10^{n-1}, 10^{n-1})$ sends the packet
    to $(00^{n-1}, 10^{n-1}, 00^{n-1})$
    and to $(00^{n-1}, 00^{n-1}, 10^{n-1})$;
  $(10^{n-1}, 10^{n-1}, 10^{n-1})$ sends the packet
    to $(10^{n-1}, 10^{n-1}, 00^{n-1})$
    and to $(10^{n-1}, 00^{n-1}, 10^{n-1})$;
parend;

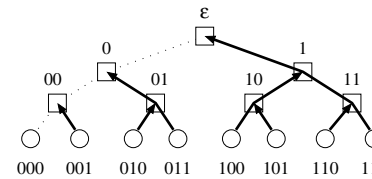


Definition 1. *For any* $a \in \cup_{i=1}^{\infty} \mathcal{B}^i$ *and* $b \in \mathcal{B}$, *define function* $\kappa(ab) = del\_trailing(a, b)$, *where*

$$del\_trailing(a, b) = \begin{cases} del\_trailing(a', b) & \text{if } a = a'b, \\ a & \text{otherwise;} \end{cases}$$

Procedure $\mathbf{C}(n)$. It is applied in the situation where all the mesh nodes have been informed. Since each mesh node is a leaf of exactly 1 x-tree, 1 y-tree, and 1 z-tree, all remaining tree nodes in all x-, y-, and z-trees are informed in a single round. Since the function $\kappa()$ is injective on nonzero strings with length n, every internal tree node $w = (x, y, z)$, $|x| < n$, $|y| = |z| = n$, of an x-tree, is informed by exactly one mesh node. $v = (\kappa^{-1}(x), y, z)$. Similarly, for y-trees and z-trees.

procedure $\mathbf{C}(n)$:
for all $x, y, z \in \mathcal{B}^n$ **do_in_parallel**
(x, y, z) sends the packet
to $(\kappa(x), y, z)$ **if** $x \neq 0$
and to $(x, \kappa(y), z)$ **if** $y \neq 0$
and to $(x, y, \kappa(z))$ **if** $z \neq 0$;



Lemma 2. *Let $b_2(n)$ ($b_3(n)$) denote the lower bound on the number of rounds of an all-output-port OAB in $M3T_n$ if the source has degree 2 (3, resp.). Then*

$$b_3(n) = b_2(n) = \tfrac{3}{2}(n+1) \text{ for odd } n,$$
$$b_3(n) = \tfrac{3}{2}n + 1 \text{ and } b_2(n) = b_3(n) + 1 = \tfrac{3}{2}n + 2 \text{ for even } n.$$

Theorem 3. *Algorithm* $\mathbf{OAB}(n, s)$ *performs the OAB in all-output-port WH $M3T_n$ using the shortest-path routing. Its number of rounds equals the lower-bound if n is odd or if n is even and s_j is a mesh node or tree root. In the remaining cases, it needs one additional round.*

We have proposed a simple OAB algorithm for all-output-port WH 3-D meshes of trees. Depending on the parity of n and the position of the source node, its complexity matches the lower bound or one more round is needed. It is interesting to compare this algorithm with the OAB algorithm in all-output-port WH **2-D** meshes of trees [4]. The 2-D case is more complicated.

It is an interesting research problem to generalize this result in several ways: (1) to *rectangular* 3-D meshes of trees; (2) to k-ary 3-D meshes of trees, $k \geq 3$; (3) to h-dimensional meshes of trees, $h \geq 4$.

References

1. D. Barth. An algorithm of broadcasting in the mesh of trees. In *CONPAR 92–VAPP*, volume 634 of *LNCS*, pages 843–844. Springer-Verlag, 1992.
2. J.-Y. L. Park and H.-A. Choi. Circuit-switched broadcasting in torus and mesh networks. *IEEE Trans. on Parallel and Distributed Systems*, 7(2):184–190, 1996.
3. P. Salinger and P. Tvrdík. Optimal broadcasting and gossiping in one-port wormhole meshes of trees. In *PDCS'99*, volume 2, pages 713–718. Acta Press, 1999.
4. P. Salinger and P. Tvrdík. Optimal broadcasting in all-port meshes of trees with distance-insensitive routing. In *IPDPS'00*, pages 353–358. IEEE CS Press, 2000.
5. Y.-C. Tseng. A dilated-diagonal-based scheme for broadcast in a wormhole-routed 2D torus. *IEEE Trans. on Computers*, 46(8):947–952, 1997.

Probability-Based Fault-Tolerant Routing in Hypercubes

Jehad Al-Sadi[1], Khaled Day[2], Mohamed Ould-Khaoua[1]

[1]Computer Science Department, Strathclyde University, Glasgow, U.K.
{jehad, mohamed}@cs.strath.ac.uk
[2]Computer Science Department, Sultan Qaboos University, Muscat, Sultanate of Oman
kday@squ.edu.om

Abstract. This paper describes a new fault-tolerant routing algorithm for the hypercube, using the concept of probability vectors. Each node calculates a probability vector whose k^{th} element represents the probability that a destination node at distance k cannot be minimally reached due to a faulty node or link. A performance comparison with the recently-proposed safety vectors algorithm, through extensive simulation, shows that the new algorithm exhibits superior performance in terms of routing distances and percentage of reachability.

1 Introduction

Efficient interprocessor communication is the key to good system performance in interconnection networks. As the network size scales up the probability of processor and link failure also increases. It is therefore essential to design fault-tolerant routing algorithms that allow messages to be routed between non-faulty nodes in the presence of faulty links and nodes. Several fault-tolerant routing strategies for the hypercube have been proposed in the literature.

The main challenge is to devise a simple and effective way of representing limited global fault information that allows optimal or near-optimal routing. Recently, there has been a number of attempts to design limited-global-information-based algorithms for the hypercube [1], [2], [3], [4]. This paper proposes a new limited-global-information-based routing algorithm for the hypercube. Routing at each node A is based on a calculated probability vector $P^A = (P_1^A, \ldots, P_n^A)$. P_k^A represents the probability that a destination node at distance k cannot be reached from node A using a minimal path due to a faulty node or link along the path. A performance comparison against the safety vectors algorithm through extensive simulation experiments is then presented. The results reveal that the new routing algorithm outperforms the safety vectors algorithm in terms of routing distances and percentage of reachability.

2 The Proposed Fault-Tolerant Routing Algorithm

The label of a node A in n-cube is written $a_n a_{n-1} \ldots a_1$, where $a_i \in \{0, 1\}$ is the bit at i-th dimension. The neighbour of a node A along the i-th dimension is denoted $A^{(i)}$.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 935-938, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The *Hamming distance* between a node A and a node B is denoted $H(A,B)$. A path between two nodes A and B is a *minimal path* if its length is equal to $H(A,B)$.

With respect to a given destination node D, a neighbour $A^{(i)}$ of node A is called a *preferred neighbour* for the routing from A to D if the i-th bit of $A \oplus D$ is 1. We say in this case that i is a *preferred dimension*. Neighbours other than preferred neighbours are called *spare neighbours*. Routing through a spare neighbour increases the routing distance by two over the minimum distance.

The basic idea of this algorithm is for each node A to determine its faulty set F_A of faulty or unreachable neighbours, to use this unsafety set to calculate its probability vector P^A, and then perform fault-tolerant routing using P^A.

Definition 1: *The faulty set* F_A of a node A is defined as

$$F_A = \bigcup_{1<i<n} f_A^i \text{, where } f_A^i \text{ is given by } f_A^i = \begin{cases} \{A^{(i)}\} & \text{if } A^{(i)} \text{ is faulty} \\ \phi & \text{Otherwise} \end{cases}$$

The kth element P_k^A of P^A denotes the probability that a destination at distance k from A is not minimally reachable from A. Since node A has $|F_A|$ faulty or unreachable immediate neighbours, and only one of the n edges incident from A constitutes a minimal path to a specific destination at distance one, the probability P_1^A is:

$$P_1^A = \frac{|F_A|}{n} \tag{1}$$

In order to compute the other elements P_k^A, $k \geq 2$, let $R_k^{A^{(i)}}$ be the probability that a destination at distance k from A is minimally reachable via its neighbour $A^{(i)}$. The probability P_k^A, $k \geq 2$, can then be expressed as

$$P_k^A = \prod_{i=1}^{n} (1 - R_k^{A^{(i)}}) \text{, where} \tag{2}$$

$$R_k^{A^{(i)}} = \begin{cases} 0 & \text{if node } A^{(i)} \text{ is faulty} \\ \frac{k}{n}(1 - P_{k-1}^{A^{(i)}}) & \text{otherwise} \end{cases} \tag{3}$$

When a node A has to forward a message M towards its destination D it applies the probability-based routing algorithm PB_Routing, outlined in Fig 1. If we can route through a preferred neighbour, $A^{(i)}$, then the associated least expected routing distance is calculated as follows:

$$Pr = h(1 - P_{h-1}^{A(i)}) + (h+2)P_{h-1}^{A(i)} \tag{4}$$

where $P_{h-1}^{A(i)}$ is the probability that a minimal path via the preferred neighbour $A^{(i)}$ to a destination at distance h is faulty. On the other hand, if we route through a spare neighbour, $A^{(j)}$, then the least expected routing distance is calculated as follows:

$$Sp = (h+2)(1-P_{h+1}^{A(j)}) + (h+4)P_{h+1}^{A(j)} \quad (5)$$

```
Algorithm PB_Routing (M: message; A,D: node)
 /* called by node A to route the message M towards its destination node D */
  if D is a reachable neighbour then deliver M to D; exit; /* destination reached */
  h = Hamming distance between A and D
  Let A(i) be a reachable preferred neighbour with least P_{h-1}^{A(i)} value;
  Pr = h(1 - P_{h-1}^{A(i)}) + (h+2)P_{h-1}^{A(i)} ; /* least expected distance through A(i) */
  Let A(j) be a reachable spare neighbour with least P_{h+1}^{A(j)} value;
  Sp= (h+2)(1 - P_{h+1}^{A(j)}) + (h+4)P_{h+1}^{A(j)} ; /* least expected distance through A(j) */
  if ∃ A(i) and ( (∃ A(j) and Pr ≤ Sp) or (~∃ A(j)) ) then send M to A(i) ;
  else if ∃ A(j) and ( (∃ A(i) and Pr > Sp) or (~ ∃ A(i)) ) then send M to A(j) ;
      else failure        /* destination unreachable */
```

Fig. 1. Outline of the proposed probability-based fault-tolerant routing algorithm

3 Performance Comparison

This section reports results from simulation experiments comparing the performance of the proposed routing algorithm to that of the safety vectors algorithm [3]. To this end, a simulation study has been conducted for both the proposed PB_Routing algorithm and safety vectors approaches over an 8-dimensional hypercube (256 nodes) with different random distributions of faulty nodes. We started with a non-faulty hypercube. The number of faulty nodes was then increased gradually up to 75% of the hypercube size with random fault distributions. A total of 10,000 source-destination pairs were selected randomly during each run.

Let *Total* be the total number of generated messages, *Delivered* be the number of delivered messages, and *FailCount* be the number of routing failure cases.

We propose the following three performance measures as the basis for comparing the safety vectors and the probability vectors algorithms.

- Percentage of unreachability $= \frac{Fail\_Count}{Total} \times 100$
- Average deviation from optimality $= \frac{1}{Delivered} \Sigma \frac{Routing\_Distance - Hamming\_Distance}{Hamming\_Distance}$

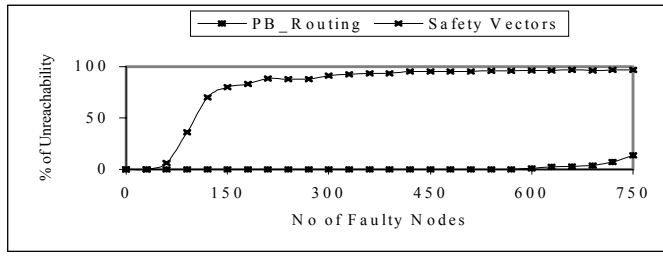


Fig. 2. Percentage of Unreachability of PB_Routing and Safety Vectors Algorithms

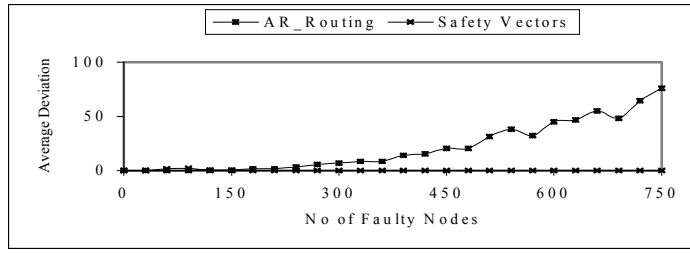


Fig. 3. Average Deviation from optimality of PB_Routing and Safety Vectors algorithms

The percentage of unreachability measures the percentage of messages that the algorithm fails to deliver to destination due to faulty components. The average deviation from optimality indicates how close the achieved routing is to the minimal distance routing. The obtained results (Fig.2, 3) reveal that PB_Routing achieves much higher reachability with low to moderate deviation from optimality.

4 Conclusion

We have proposed a new probability-based fault-tolerant routing algorithm for the hypercube. Each node A calculates a numeric probability vector P_A and uses it in performing fault-tolerant routing. The performance of the proposed algorithm has been compared to that of the recently proposed safety vectors algorithm through extensive simulation. The new algorithm outperforms the safety vectors algorithm in terms of reachability and deviation from optimality.

References

1. Chiu, G.M., Chen, K.-S.: Fault-tolerant routing strategy using routing capability in hypercube multicomputers, Proc. Int'l Conf. Parallel and Distributed Systems, pp. 396-403, 1996.
2. Lee, T.C.: J.P. Hayes, A fault-tolerant communication scheme for hypercube computers, IEEE Trans. Computers, vol. 41, no. 10, pp. 1,242-1,256, Oct. 1992.
3. Wu, J.: Adaptive fault-tolerant routing in cube-based multicomputers using safety vectors, IEEE Trans. Parallel and Distributed Systems, vol. 9, no. 4, pp. 321-334, April 1998.
4. Wu J., Fernandez, E.B.: Broadcasting in faulty hypercubes, Proc. 11th Symp. Reliable Distributed Systems, pp. 122-129, 1992.

Topic 14
Instruction-Level Parallelism and Processor Architecture

Kemal Ebcioglu

Global Chair

This year, the Euro-Par conference is being held in beautiful Munich, Germany. I am very honored to welcome you to the instruction level parallelism and processor architecture sessions of Euro-Par 2000!

Instruction level parallelism (ILP) and processor architecture are important and growing fields. ILP research aims to extract very fine-grained parallelism not only from scientific code, but also from the irregular, general code that pervades applications and operating systems. Thus, successful ILP techniques can have a paramount effect on the performance of the entire computer system. ILP researchers in the academia and industry have continuously been designing leading-edge techniques to increase processor performance, such as micro-architecture enhancements, memory latency tolerance, and aggressive compiler algorithms. Samples of these techniques can be seen among the present collection of papers.

This year, 29 papers were submitted to the ILP and processor architecture topic. The selection process was very competitive, and difficult for the topic organizers. At the end, 4 submissions were accepted as regular papers, and 8 were accepted as short papers.

I would like to thank the other members of the organizing committee of the present topic, namely Prof. Theo Ungerer (the Local Chair), Prof. MariaGiovanna Sami, and Prof. Nader Bagherzadeh (the Vice-Chairs), who painstakingly reviewed many papers and provided insightful remarks. Prof. Ungerer took the lead in the publicity effort, set up an excellent Web-based review database for the papers, and at the end represented us during the Euro-Par program committee meeting. I am also grateful to our referees for lending us their expertise and providing rigorous reviews. I would finally like to thank the Euro-Par 2000 conference organizers for their continued support.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 939–939, 2000.
© Springer-Verlag Berlin Heidelberg 2000

On the Performance of Fetch Engines Running DSS Workloads*

Carlos Navarro, Alex Ramírez, Josep-L. Larriba-Pey, and Mateo Valero

Universitat Politècnica de Catalunya
Jordi Girona 1–3, D6
08034 Barcelona (Spain)
{cnavarro, aramirez, larri, mateo}@ac.upc.es

Abstract This paper examines the behavior of current and next generation microprocessors' fetch engines while running Decision Support Systems (DSS) workloads. We analyze the effect of the latency of instructions being fetched, their quality and the number of instructions that the fetch engine provides per access. Our study reveals that a well dimensioned fetch engine is of great importance for DSS performance, showing gains over 100% between a conventional fetch engine and a perfect one. We have found that, in many cases, the I-cache size bounds the benefits that one might expect from a better branch prediction.
The second part of our study focuses on the performance benefits of a code reordering technique for the Database Management System (DBMS) that runs our DSS workload. Our results show that the reordering has a positive effect on the three parameters and can speed-up the DSS execution by 21% for a 4 issue processor, and 27% for an 8 issue one.

1 Introduction

Fetch engine performance is characterized by three different factors: the latency of instructions being fetched from memory, the number of instructions fetched per access, and the quality of those instructions.

Instruction latency is caused by the speed gap between processing and memory. While the pipeline needs new instructions every cycle, the memory can only supply them at a ratio of tens or even hundreds of cycles. Reducing the I-cache misses has been addressed using software and hardware techniques. On the software side we have code reordering techniques, like [7,11]. On the hardware side we have set associative caches, hardware instruction prefetching [17], and victim caches [8], among others.

The number of instructions fetched per cycle, fetch bandwidth, is a problem when more than one basic block has to be provided per cycle. Bandwidth is limited by the execution of non-contiguous instructions and the hardware width

* This research has been supported by CICYT grant TIC-0511-98, the Spanish Ministry of education grant PN98 43443683-1 (Carlos Navarro), the Generalitat de Catalunya grant 1998FI-003060-26 (Alex Ramirez) and CEPBA.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 940–949, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of the fetch engine. The execution of non-contiguous basic blocks has been addressed also, both in hardware and software. The hardware solutions comprise the branch address cache [20], the collapsing buffer [5], and the trace cache [10,15]. Also, in our previous work [12,13] we have addressed this problem with a software code reordering technique, the Software Trace Cache (STC), and the interaction between the software and hardware trace cache [14].

The quality of instructions is determined by branch prediction accuracy. Dynamic branch prediction schemes have been used during the last years to avoid stopping the fetch until branch resolution [16]. But the use of this prediction mechanisms also introduces wrong path instructions in the pipeline whenever a branch is mispredicted. A lot of work has been done during the last years to increase branch prediction accuracy [6,16,21].

DBMSs are highly structured codes, perform many procedure calls and have plenty of control statements to handle all types of error situations. This causes a great level of control flow activity in the code and larger instruction working sets when compared with other common integer codes[1,2]. Recent work has shown that those instruction working sets can be a problem for the instruction cache sizes used in current generation microprocessors [1].

Our results show that for DSS workloads, the fetch engine has a large influence on the IPC. In Figure 1 we compare the performance of a conventional fetch engine against the best fetch engine possible for 4 and 8 issue superscalars. This perfect fetch engine returns the full width of the next correct path instructions every cycle. That is, it does not have latency problems, it uses all the possible bandwidth, and it uses a perfect branch prediction. We can see that the perfect fetch engine shows speedups of 98% and 59% over the 16KB and 32KB engines for the 4 issue processor and of 116% and 70% for the 8 issue processor.

With all this evidence, we study how DBMSs exercise the different parts of the current processors' fetch engines, and how different improvements on the fetch engine can help their execution. We analyze the fetch impact on the overall performance, the isolated behavior of the three main fetch parameters, and their effect on performance. We also analyze the effects of a code reordering technique on each fetch engine component and its effect on performance.

The rest of the paper is structured as follows. Section 2 presents our experimental setup. Sections 3, 4, and 5 study the isolated effect on performance of fetch latency, quality and bandwidth. Section 6 addresses the effects of applying a code reordering scheme to our workload. Finally we conclude in Section 7.

2 Experimental Setup

Our DSS workload runs on top of the PostgresSQL 6.3 DBMS [18]. Our workload is modeled with a session that executes queries 3 and 17 of the Transaction processing council TPC-D benchmark [19].

We set up the Alpha version of the Postgres 6.3 database on digital Unix v4.0 compiled with the -O3 flags of the Digitals's C compiler. A TPC-D database is used with scale factor 0.1 (100MB of raw data).

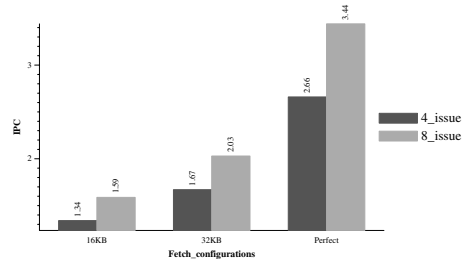


Fig.1. Comparison between two conventional fetch engines and the perfect one for 4 and 8 issue superscalar processors. The experiment shows results for cache sizes of 16KB and 32KB. The conventional fetch engine uses a 2048 entries Bimodal predictor.

The simulator used in this study is based on the sim-outorder timing simulator included in the Simplescalar-Alpha v3.0 tool set [4]. Table 1 shows the configuration we have used for our baseline architecture. The I-cache and branch prediction setups are varied across the study. Since the complete execution requires over a billion instructions, we have used sampling in order to reduce simulation times. Simulations are performed using the following sequence: 50 Million detailed simulation sample followed by a 150 Million fast simulation sample where we just emulate the ISA.

We model a branch misprediction in the following way. When a branch misprediction is detected, the execution engine spends a fixed number of cycles arranging the window and other internals before re-starting the fetch in the correct path. This number of cycles is what we call *misprediction recovery penalty* and it is a simulation parameter. After those cycles, fetch is re-started and the issue of correct path instructions restarts 2 cycles later[9].

In this work we are using two different processor widths, 4 and 8. The fetch engine for the 4 issue processor predicts a single branch per cycle, while for the 8 issue processor it can predict a second branch if the first was not taken.

3 Effect of Instruction Latency on Performance

The first parameter we explore is the latency of the instructions being fetched from memory, and the impact that it has on the overall performance. Our analysis will be structured as follows: First we will show the perceived latency for

Parameters	Values	Parameters	Values
Core		Misprediction recovery penalty	1
Issue width	4,8	*Memory*	
Window size	128	Inst L1	several sizes/ 32 byte/ 2 way
L/S queue size	128	Data L1	64K/ 32 byte/ 2 way
FUs	Unbound fully pipelined	Combined L2	2MB/ 64 byte/ 2 way
Branch prediction		Latencies	1/ 7 / 80
Direction	Several schemes	Instruction TLB	16 4K entries/ 4 way
Address	BTB 4K entries/ 4 way	Data TLB	32 4K entries/ 4way
RAS	128 entries		

Table1. Baseline processor

several cache sizes, and finally we will show how those latencies affect the overall execution performance.

Given that with the current technology trends our L2 latency could be quite aggressive, we will also provide results for a L2 latency of 15 cycles. This latency is more feasible for future processor clock speeds, even with integration of L2 in the processor die. For example, in [3], where a 1GHz processor with integrated 2MB L2 cache is evaluated, the latencies for L2 are larger than 15 cycles.

Our metric for the perceived instruction latency, will be the Average Memory Access Time for instructions ($AMAT_i$). This metric will show the average latency that our fetch engine perceives per instruction. $AMAT_i$ is computed as: $Icache\ miss\ rate * L2\ latency + (Icache\ miss\ rate * L2\ instruction\ miss\ rate) * memory\ latency$. With an $AMAT_i$ equal to 1, the fetch engine would never block waiting for instructions.

Figure 2(a) shows the $AMAT_i$ for several I-cache sizes when we use both 7 and 15 cycles L2 latencies. We observe that with an 8KB I-cache the fetch engine must wait more than 1.5 cycles per instruction or even 2.08 cycles for a 15 cycles L2 latency. We can also see that for 15 cycles, even a 64KB I-cache shows a latency that still is far from the desired 1. Our results show that the working set starts to fit in caches with sizes of 128KB or larger. This is not strange because the instruction footprint of our queries is larger than 230KB.

Figure 2(b) shows the IPC for different I-cache sizes using perfect branch prediction for 7 cycles L2 latency. The perfect I-cache shown in that Figure is used as an upper bound. Results are shown for both 4 and 8 issue processors. We can see that the decrease in performance for I-caches smaller than 128KB is important. This is due to the increase in exposed latency and is even more evident if we focus on the 8 issue processor. For a 4 issue processor and caches of 8 and 16KB it is better to double I-cache size than to double the issue width.

Figure 2(c) shows results for a 15 cycles L2 latency. We can see that the slope for both 4 and 8 issue is steeper than the one of Figure 2(a). In this case, the drops in IPC are significant even for large caches. For instance, we have speedups of 18% and 23% in 4 and 8 issue processors when we go from a 64KB to a 128KB I-cache.

4 Effect of Instruction Quality on Performance

In this section we want to quantify if the branch prediction accuracy has the same importance as the I-cache performance. In order to inspect that, we have tested the behavior of several branch predictors. In particular we have used Bimodal and Gshare branch predictors of 10, 12, 14, 16, and 18 bits and Hybrid branch predictors of 12 and 14 bits.

The branch prediction accuracy of those predictors is between 98.34% and 88.55%. In general, the branch prediction accuracy is good, compared to the prediction rates obtained on several SpecInt95 benchmarks [6].

Figure 3(a) shows the IPC we can expect from a 4 issue processor running the branch predictors we have tested. Results show an interesting effect. On one

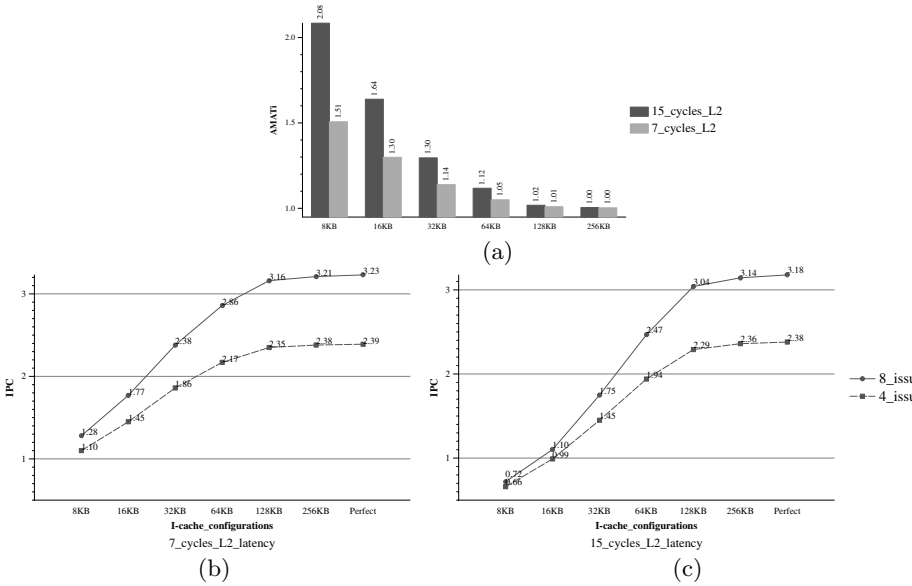


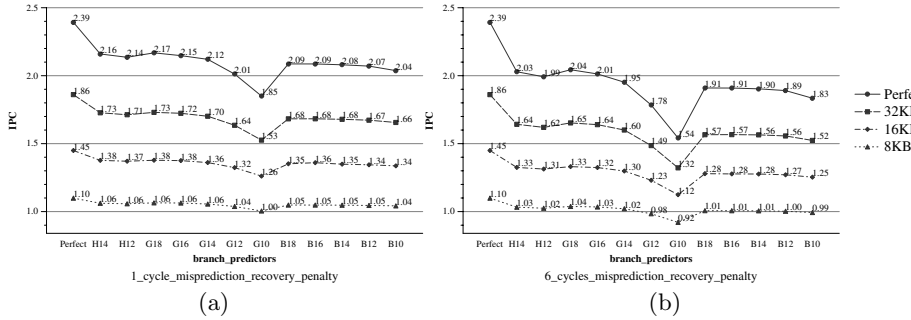
Fig.2. (a) AMATi in cycles for different L1 I-cache sizes and L2 latencies. Charts (b),(c): IPC for several I-cache configurations. Chart (b) shows results for 7 cycle L2 latency. Chart (c) shows results for 15 cycle L2 latency.

hand, the difference in branch prediction accuracy between the worst performing branch predictor (G10) and the oracle one are clearly reflected in the IPC for the results with a perfect I-cache. That is, the quality of instructions directly enhances the overall performance. On the other hand, for the system with an 8KB I-cache, the benefits of the same increase in instruction quality are very small. We can see that the impact on IPC of the increasing branch prediction accuracy grows when we increase the I-cache size. So the perceived latency is hiding the benefits of the increase in quality. Thus, for our workloads, spending a lot of hardware in complex branch predictors is not cost effective unless AMAT_i is small enough.

The results we have seen so far are for simulations where the misprediction recovery penalty was set to 1. This leads to small branch misprediction penalties. One question that could arise is, how the effect of latency over instruction quality is modified by an increasing branch misprediction penalty.

Figure 3(b) shows results for branch misprediction recovery cycles equal to 6 leading to penalties of at least 8 cycles. In that Figure, we can see that, the effect observed in the last section still exists when we have larger branch misprediction penalties. For branch misprediction recovery of 6 cycles it is still better to duplicate an 8KB I-cache than to have a perfect branch predictor.

If we have a design with a misprediction recovery time of 12 cycles (chart found at [9]), it is always better to go to the perfect branch prediction than to duplicate the cache size. However, it is interesting to note that I-cache size is still



(a) (b)

Fig.3. IPC for each of the branch predictors tested. Charts (a) is for misprediction recovery penalties of 1 cycles, while chart (b) is for misprediction recovery penalties of 6 cycles. All simulations are for a 4 issue processor and L2 latency of 7 cycles.

having a bounding effect on the performance benefits of better branch prediction accuracy.

5 Effect of Fetch Bandwidth on Performance

Instruction fetch bandwidth will be a major limiting factor to high performance in next generation microprocessors. Nevertheless, fetch bandwidth could be also a problem for 4 or 8 issue processors.

Several experiments have been performed to characterize the impact that fetch bandwidth has on the performance. In particular we have tested the impact of increasing the fetch bandwidth for a 4 issue processor to 8 instructions per cycle. The complete analysis can be found at [9] and has not been shown here due to space limitations.

Our results show that for 4 and 8 issue processors, large increases in bandwidth do not reflect in such improvements in IPC. Thus, for our workload and issue widths, increasing the fetch bandwidth is not critical. Instead, its is more important to fetch the right path instructions.

6 Code Reordering

In the previous sections we have shown the high impact that the fetch engine design can have in the overall execution performance of our DSS workload. From this evidence, it seems that for our workloads, it can be interesting to use a supplementary technique in order to help I-caches. This technique can be a code reordering scheme.

Code reordering techniques have been used for a long time in order to reduce the I-cache misses [11]. In our previous work, [12,13,14] we have shown that DSS workloads are a good target for these techniques. In particular we presented a reordering technique, the Software Trace Cache (STC), aimed at increasing the

fetch bandwidth for future aggressive wide superscalars. This technique has been proven as efficient increasing the number of non taken branches and reducing the I-cache misses.

In this section we want to characterize how the application of the STC reordering affects each of the fetch parameters and how the overall performance can benefit from it. In order to do so, we have used the same PC translation mechanisms used in [13] to simulate the reordering, and we have performed simulations for three different I-cache sizes: 8, 16 and 32 KB.

First, we will study the impact of the STC reordering on the latency. Figure 4(a) shows the $AMAT_i$ for both reordered and non reordered workloads. We can see that for the same I-cache size, the reordered workload always shows a lower latency. We can see that for a 32KB I-cache, the reordered workload has an $AMAT_i$ of the same order as a 64KB one with a non reordered workload.

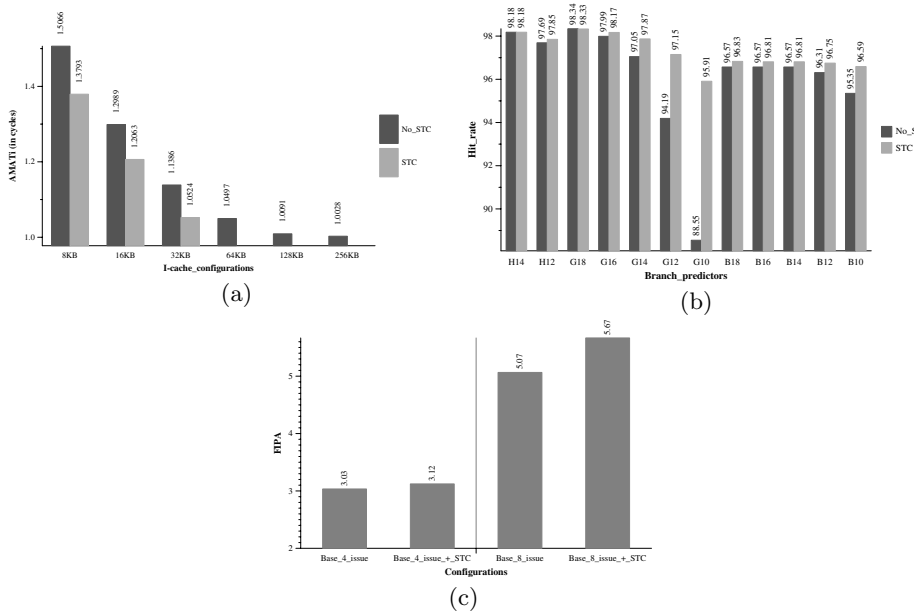


Fig.4. (a) $AMAT_i$ for both reordered and non reordered workload. Results for STC are shown up to 32KB I-cache. (b) Branch prediction hit rates for both reordered and non reordered workload. The branch predictors are the same used in Section 4. (c)Fetched Instructions per Access (FIPA) for both reordered and non reordered workload. The 2 leftmost bars are for 4 issue, while the right ones are for 8 issue.

The second aspect we want to analyze is the impact that the STC has in the quality of the fetched instructions. Figure 4(b), shows the branch prediction hit rate for both reordered and non reordered workloads, for the same branch predictors used in Section 4. Results show that the branch prediction accuracy improves always. This is not strange because the STC reorders the basic blocks in

order to leave the biased branches in a not taken form. This leads to a reduction in the destructive conflicts in the branch prediction tables.

The third parameter we want to analyze is the effect that the reordering has in the effective fetch bandwidth. This effect has been shown in [12,13] for aggressive superscalar processors. Here we want to study it for our 4 and 8 issue superscalars. Figure 4(c) shows the number of fetched instructions per access to the fetch engine (FIPA) for the baseline engine on the 4 and 8 issue superscalar processors. In this simulations the branch prediction scheme used is the H12 and the I-cache evaluated is 32KB.

The results show that, while the increase in bandwidth for 4 issue superscalar is small, the increase for 8 issue is bigger (about 12%). This fact, coupled with the branch prediction accuracy results, means that we are not only fetching more instructions but more accurate instructions.

Finally we want to see the overall effect that our STC technique has over the execution performance. Figure 5 shows the performance increases that a conventional fetch engine would expect if the code was reordered with the STC. Results show that for the three configurations we have tested with STC, we always have similar performance to a system with double I-cache size.

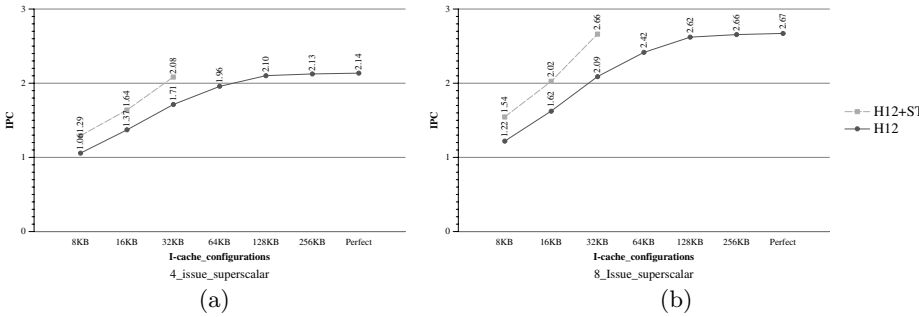


Fig.5. Comparison between the IPC of a non-reordered and a STC reordered workload. Chart (a) show results for 4 issue while chart(b) shows results for 8 issue. Bottom line shows the IPC for the baseline issue processor with the H12 branch predictor and several cache sizes. Upper line shows the IPC of the same configuration if the code is reordered using the STC.

7 Concluding Remarks

In this paper we have analyzed the high performance impact that the fetch engine behavior has on the overall performance of a current superscalar processor while running DSS workloads. We have seen that a bad tuning of the engine can lead to high penalties in the IPC.

The impact of the three main parameters of the fetch engine has been studied. In particular we have seen that, due to the size of the working set of our workload,

the impact of I-cache misses is the most important. The fetch engine can not tolerate the latencies of L2 caches. This effect can even hide the benefits of a better branch predictor. So, for these workloads it is more important to spend chip area in hiding L2 cache latency (bigger I-cache for example) than adding a more complex branch predictor.

Finally, we have shown how a code reordering technique (STC) affects the behavior of the different fetch parameters. It is specially interesting the fact that the branch prediction accuracy is improved without extra hardware cost. Also, we have seen that the STC can speed-up the DSS execution in more than 21% for a 4 issue processor, and 27% for an 8 issue one.

References

1. A. Ailamaki, D. Dewitt, M. Hill, and D. Wood. Dbms on modern processors: Where does time go? *Proc. of the 25th Int. Conf. on Very Large Data Bases*, 1999.
2. L. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. *Proc. of the 25th Intl. Symp. on Comp. Architecture*, 1998
3. L. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of chip-level integration on performance of oltp workloads. *Proc. of the 6th Intl. Conf. on High Performance Comp. Architecture*, January 2000.
4. D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. *Technical Report TR-1308, Comp. Sciences Dept., Univ. of Wisconsin-Madison*, 1996.
5. T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proc. of the 22th Intl. Symp. on Comp. Architecture*, pages 333–344, June 1995.
6. T . Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. *Proc. of the 32th Intl. Symp. on Microarchitecture*, 1999.
7. W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proc. of the 16th Intl. Symp. on Comp. Architecture*, 1989.
8. N. J. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. of the 17th Intl. Symp. on Comp. Architecture*, pages 364–373, June 1990.
9. C. Navarro, A. Ramírez, J. Larriba-Pey, and M. Valero. Fetch Engine Design Decissions for DSS Workloads *Tech. report UPC-DAC-2000-9* , Feb. 2000.
10. A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized arround trace segments independent of virtual address line. *U.S. Pat. 5.381.533*, Jan. 1995.
11. K. Pettis and R. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN'99 Conf. on Programming Language Design and Implementation*, pages 16–27, 1990.
12. A. Ramírez, J. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. *Proc. of the Intl. Conf. on Parallel Processing*, pages 238–245, September 1999.
13. A. Ramírez, J. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *Proc. of the 13th Intl. Conf. on Supercomputing*, pages 119–126, 1999.
14. A. Ramirez, J. Larriba-Pey, and M. Valero. Trace cache redundancy: Red & blue traces. *Proc. of the 6th Intl. Conf. on High Performance Comp. Architecture*, 2000.
15. E. Rottenberg, S. Benett, and J. E. Smith. Trace cache: a low latency aprroach to high bandwith instruction fetching. *Proc. of the 29th Intl. Symp. on Microarchitecture*, pages 24–34, December 1996.

16. J. E. Smith. A study of branch prediction strategies. *Proc. of the 8th Intl. Symp. on Comp. Architecture*, 1981.
17. J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. *Supercomputing 92*, pages 588–597, November 1992.
18. M. Stonebreaker and G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, October 1991.
19. Transaction Processing Performance Council (TPC). Tpc benchmark d (decision support) http://www.tpc.org. Standard Specification, Revision 1. 2. 3, 1993–1997.
20. T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proc. of the 7th Intl. Conf. on Supercomputing*, pages 67–76, July 1993.
21. T. Y. Yeh and Y. N. Patt. Two-level adaptive branch prediction. *Proc. of the 24th Intl. Symp. on Microarchitecture*, pages 51–61, 1991.

Cost-Efficient Branch Target Buffers

Jan Hoogerbrugge

Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
jan.hoogerbrugge@philips.com

Abstract. Branch target buffers (BTBs) are caches in which branch information is stored that is used for branch prediction by the fetch stage of the instruction pipeline. A typical BTB requires a few kbyte of storage which makes it rather large and, because it is accessed every cycle, rather power consuming. Partial resolution has in the past been proposed to reduce the size of a BTB. A partial resolution BTB stores not all tag bits that would be required to do an exact lookup. The result is a smaller BTB at the price of slightly less accurate branch prediction. This paper proposes to make use of branch locality to reduce the size of a BTB. Short-distance branches need fewer BTB bits than long-distance branches that are less frequent. Two BTB organisations are presented that use branch locality. Simulation results are given that demonstrate the effectiveness of the described techniques.

1 Introduction

Branch target buffers (BTBs) play an important role in many pipelined processors in which branch prediction is employed to provide a steady flow of instructions in the presence of branch instructions [1]. A BTB is a cache in which branch targets are stored. In its most elementary form, a BTB is a table indexed by the lower part of the pc (program counter) with entries consisting of a tag and a branch target. In the fetch stage of the pipeline the BTB is indexed by the lower part of the pc. The tag of the returned entry is compared with the remaining bits of the pc. If these pc bits match, the instruction that is fetched from the instruction cache will be a branch instruction. In that case the target returned by the BTB will be assigned to the pc so that instructions will be fetched from that address in the next cycle. This scheme works fairly well, since most branches are taken (65-70% on average) and branches are stored in the BTB when they are taken.

A first improvement on this scheme is to make the BTB set-associative to reduce conflicts between branches mapping onto the same entry. Instead of indexing one entry from the BTB, a set of n entries is indexed and n parallel tag compares determine the presence of a branch and, if present, which entry it contains.

A further improvement is to predict the direction, i.e. taken or not taken, of a branch not by the presence of it in the BTB but in a more sophisticated way. A common method is to include a two bit saturating counter in a BTB entry [2]. This counter is incremented on a taken branch and decremented on a not-taken branch. A branch will be predicted to be taken if its corresponding counter has its highest bit set. Very often these counters are decoupled from the BTB and stored in another table which has typically more entries than the BTB and is often accessed differently to exploit branch correlation [3, 4, 5].

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 950–959, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Another improvement is to employ a return address stack (RAS) [6]. Function return branches are indirect jumps with varying targets. This makes that the target stored in the BTB, which is simply the target of the last execution of the branch, is a worse target predictor. A RAS can improve this by letting function call branches push the return address on it and letting function return branches pop values of it and use them as target prediction. For this to work, the fetch stage must know whether branches are function calls, function returns, or neither. This sort of type information can be stored in the BTB. If the BTB reports that the fetched instruction is a function call it pushes the address of the next sequential instruction on the RAS. If the BTB reports that the fetched instruction is a function return it pops a value of the RAS and uses it for target prediction.

The objective of the work described in this paper is to improve the cost-efficiency of BTBs by using fewer bits per BTB entry. This will make BTBs smaller and reduce power dissipation. The latter can be quite high since a BTB is a table of typically a few kbyte in size that is accessed every cycle[1]. Alternatively, with the same area and power budgets one can make a BTB that delivers more performance with the techniques described in this paper.

We obtain a reduction in storage requirements by making the tag and target fields of a BTB entry smaller. Both reductions are shown in Fig. 1. Reduction of tag fields is known as partial resolution and has been proposed by Fagin and Russell [8]. The main contribution of this paper is target field size reduction. We make use of branch locality, i.e., most branch distances are short. Two schemes are presented that exploit branch locality with comparable performance. In one scheme BTB entries can store a short branch and two entries are required to store a long branch, which occurs less frequently. In the other scheme only one entry per set of a set-associative is able to store long branches. The other entries can only store short branches and are therefore smaller.

The remainder of the paper is organised as follows. Section 2 describes the simulation environment we used for our experiments. Section 3 discusses partial resolution. Section 4 presents the two BTB organisations that exploit branch locality. Section 5 concludes the paper.

2 Simulation Environment

We used the SimpleScalar v2.0 tool set for our experiments in which we modified the BTB simulation code [9]. SimpleScalar resembles the 32-bit MIPS architecture. We used a decoupled system in which the BTB detects branches and provides branch targets and type information, and a separate predictor provides direction predictions. To expose the effect of different BTB organisations, we use a fairly aggressive hybrid branch predictor consisting of a 4096 entry gshare predictor with 12 bits history, a 2048 entry bimodal predictor, and a 2048 entry meta predictor [3, 4, 5]. We used a 32 entry RAS. The BTBs used are 4-way set-associative with LRU replacement with sizes varying from 32 to 1024 entries.

[1] According to Musoll [7], the Intel Pentium Pro dissipates 5% of its total power dissipation in its 512-entry BTB. This percentage becomes higher for a less aggressive processor or a processor with less complex decoding. The percentage will also be higher for 64-bit processors.

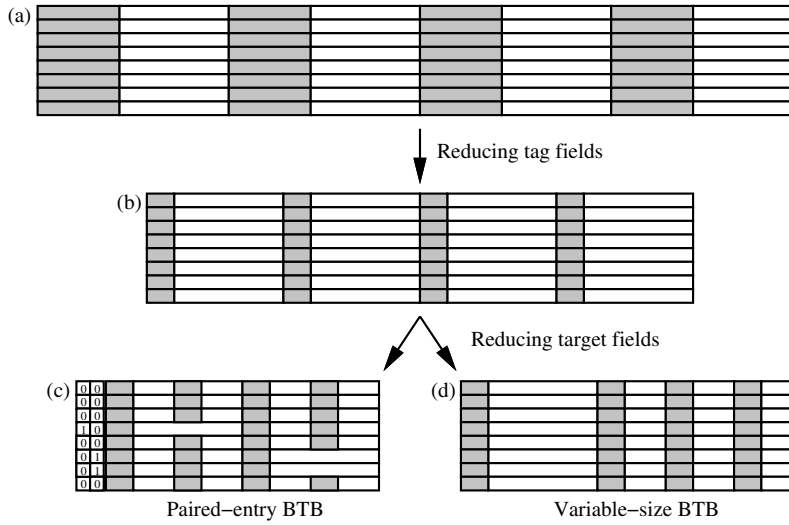


Fig. 1. Organisations of four types of BTBs. The gray parts represent tag fields while the white fields represent target and type information fields. The picture is approximately on scale. (a) shows a traditional 4-way set-associative BTB with 32 entries. (b) shows a BTB with partial resolution in which tag fields have been reduced. (c) and (d) show two organisations in which branch locality is exploited. (c) shows a paired-entry BTB (PE-BTB) in which long branches occupy two BTB entries. (d) shows a variable-size BTB (VS-BTB) in which only one BTB entry per set can store a long branch.

As a benchmark set we used the eight benchmarks from SPECint95. Simulation was limited to 100 million instructions.

3 Partial Resolution

BTBs with partial resolution do not store all the address bits that are not used for indexing in tags [8]. This reduces both the number of storage bits as well as the size of the comparators that perform the tag compare. The result of partial resolution is so-called false-hits; the BTB hits for an instruction which is not a branch instruction or is a branch instruction that does not correspond to the information returned by the BTB. If the branch predictor predicts taken, then the control flow is very likely to be directed in the wrong direction. This will be detected and corrected in a later pipeline stage. The effect is a cycle penalty comparable to the misprediction penalty. How often this occurs will depend on how many tag bits are used. Fagin and Russell conclude that 2 tag bits are necessary to obtain 99.9% of the accuracy of full resolution and 3 tag bits for 99.99%.

Our experiments however showed that more tag bits are required. Figure 2 shows the results of these experiments. We used two benchmarks with a large branch working set, *gcc* and *go*, and two benchmarks with a small branch working set, *compress* and

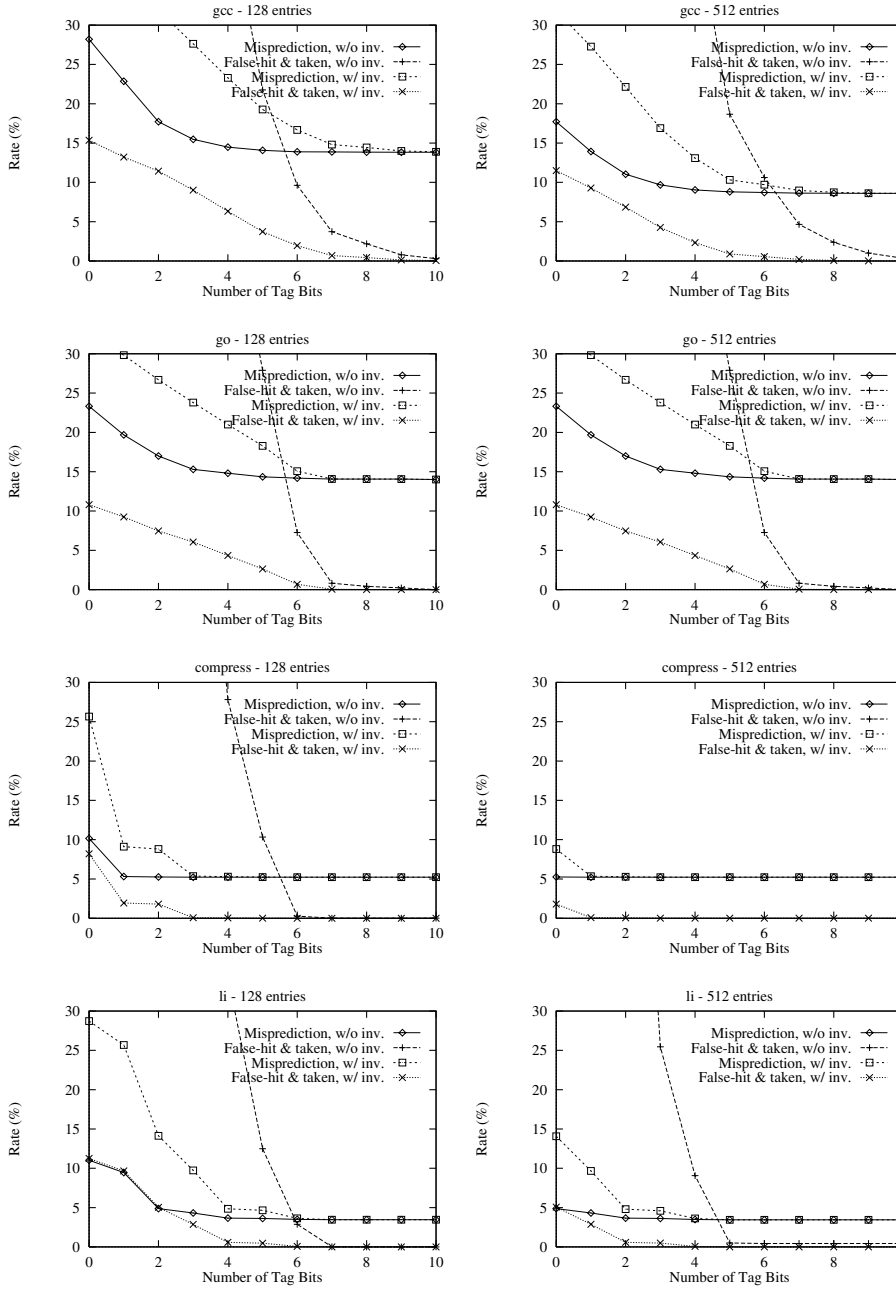


Fig. 2. Misprediction and false-hit rates for *gcc, go, compress*, and *li* for 128 and 512 entry BTBs as a function of the number of tag bits. The false-hit taken rate of *compress* for 512 entries without invalidation is so high (more than 430%) that it lies beyond the vertical range of the graph.

li. We used BTBs of 128 and 512 entries. Two policies were used to handle false-hits: invalidation and no invalidation. With invalidation a BTB entry that causes a false-hit is invalidated to prevent it continuing to cause false-hits[2]. This is achieved by a special type information value in the type information field of the BTB entry. Alternatively, one can write a random value in the tag field. The graphs of Fig. 2 show two values: the misprediction rate and the false-hit taken rate. The latter is the ratio of the number of non-branch instructions that cause a false-hit that are predicted taken relative to the number of branch instructions. Obviously, this value can be more than one when only a few tag bits are used.

Several conclusions can be drawn from the results shown in Fig. 2. First, both the misprediction and the false-hit taken rate decrease as the number of tag bits increases. More tag bits reduces the probability of branches being mixed up with other branches (which decreases the misprediction rate) and that of branches being mixed up with non-branch instructions (which decreases the false-hit taken rate). Invalidation reduces the false-hit taken rate significantly but increases the misprediction rate because BTB entries of branches are invalidated. Because the decrease in false-hit taken rate is much higher than the increase in misprediction rate, it is a good design decision to invalidate BTB entries that cause false-hits.

The number of tag bits required for a misprediction rate close to the misprediction rate for full resolution and a false-hit taken rate close to zero depends clearly on the branch working set size of the application. For the small branch working sets of *compress* and *li*, one to four tag bits will be sufficient. For the larger working sets of *gcc* and *go*, seven to eight tag bits will be required. This is significantly more than the two to three tag bits advised by Fagin and Russell. Because *gcc* and *go* are more representative of real-world applications than *compress* and *li*, we propose to use at least eight tag bits. Note that BTBs with fewer entries will have a higher false-hit rate and will therefore need more tag bits to reduce this to close to zero.

4 Exploiting Branch Locality

The tag and target fields are the largest fields of a BTB entry. Partial resolution shortens the length of tag fields. In this section we show how target fields can be shortened by making use of branch locality, i.e., the property of most programs that most branch distances are short. Branch locality is already being used for pc-relative branches in most architectures to reduce code size. Storing pc-relative offsets in the BTB is likely to be infeasible since a BTB lookup and addition would have to be performed sequentially in one clock cycle. A solution is to use page-relative addressing. Short branches are branches in which only the lowest n significant[3] pc bits change. The other branches are long branches. The idea is to choose a small value for n such that most branches are short branches and to develop a BTB such that short branches are stored in fewer

[2] Fagin and Russell do not mention invalidation in their paper [8]. It is therefore not clear whether they apply this.

[3] In many architectures instructions are aligned on 2^m byte addresses. In that case the lowest m bits are always zero and therefore not significant and do not have to be stored in the BTB.

bits than longer branches. First we determine a value for n. Figure 3 shows the percentage of short branches as a function of the number of addressing bits for pc-relative and page-relative addressing for *gcc*. Function return branches were excluded in this measurement since their targets are stored in the RAS and not in the BTB. The measurement shows that pc-relative addressing would be a little bit more effective for our purpose than page-relative addressing. The reason is that a short-distance branch across a boundary that is a high power of two is a long branch. Nevertheless, pc-relative addressing is not applicable for the reason mentioned above. The graph of Fig. 3 starts to level off at $n = 10$, at which 86% of the branches are short branches. Therefore, $n = 10$ will be a suitable value.

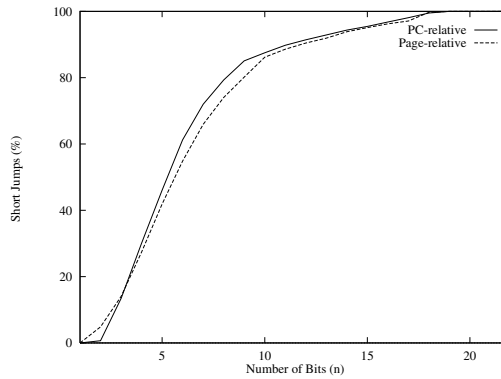


Fig. 3. Percentage of short branches as a function of the number of offset bits n. In pc-relative addressing, a branch from a to b is a short branch if $-2^{n-1} \leq b - a < 2^{n-1}$. In page relative addressing, a branch from a to b is a short branch if only the n lowest significant bits of a and b differ in value.

4.1 Paired-Entry and Variable-Size BTBs

We propose two BTB organisations for exploiting branch locality: paired-entry and variable-size BTBs.

—A *paired-entry BTB (PE-BTB)* is a set-associative BTB in with which an entry can contain one short branch and a pair of two entries from one set is needed to store a long branch. For efficient implementation, entry pairs are required to be adjacent and aligned in the set. A *pair-bit* is required per entry pair to determine whether entries are paired. For optimal resource utilisation, the total storage requirements of two short branches should be equal to the storage requirements of one long branch. This limits the range of possible tag and target sizes; the sum of 2 tags and 2 short targets should be equal to 1 tag and 1 long target (ignoring type information fields). To obtain better matching, one can decide to use different tag sizes for long and short branches or for different ways.

—A *variable-size BTB (VS-BTB)* is a set-associative BTB with which some part of the entries of each set can store both long and short branches while the remaining set entries can store only short branches.

Using the target information from the BTB is straightforward. The lookup returns on a hit whether a short or a long branch is hit. In the case of a short branch and a taken prediction, only the n lowest significant bits of the pc are updated; the upper bits retain their value. In the case of a PE-BTB, the pair-bits are used in the tag comparison. If a long branch is stored in two BTB entries, the tag compare of the entry whose tag bits are used to store the long target has to be disabled.

Some design options exist in updating the BTB. In the case of updating a PE-BTB with a long branch: (1) one can victimise the pair containing the least recently used entry, or (2) one can victimise the pair which has been least recently used, where a pair is used when one of its entries is used. To understand the difference, consider the following set of four entries showing the latest access times ($T_1 < T_2 < T_3 < T_4$):

T_1	T_4	T_2	T_3

In the case of the first alternative, the first pair would be victimised (because it contains the first entry which is the least recently used entry), whereas in the case of the second alternative the second pair would be victimised (because this pair was used least recently). We opted for the first alternative because of its simplicity and because the differences in prediction accuracy are negligible. For VS-BTBs one has to decide whether short branches are allowed in long entries or whether long entries are reserved exclusively for long branches. We chose the first alternative, which yields a slightly better prediction accuracy.

4.2 Evaluation

To evaluate the cost-efficiency of the discussed techniques we defined the four BTB types that in Table 1. All types are 4-way set-associative and use LRU replacement. In the entry size calculations we neglected the bits for the LRU administration, which are the same for all types. In the case of partial resolution, we used 8 bit tags. For short branches we used 10 bit page-relative addressing. We assumed a 32-bit RISC architecture in which instructions were 32-bit aligned, so absolute addresses were 30 bits long. Two bits of type information were used per entry, which is sufficient to distinguish four cases: (1) function call branches, (2) function return branches, (3) invalidated entries, and (4) the normal case. We varied the number of entries between 32 and 1024 in powers of two ($2^n, 5 \leq n \leq 10$).

Figure 4 shows BTB size vs. prediction accuracy curves for the four BTB types for the eight SPECint95 benchmarks. Figure 5 shows the average of the eight benchmarks. The curves obtained for the benchmarks corresponding to the four BTB types run in parallel until the BTB reaches a size at which it no longer constrains the prediction accuracy. The distances between the curves indicate the improved cost-efficiency of partial resolution, VS-BTBs, and PE-BTBs. The point at which the curves meet depends on the branch working set size of the benchmark. This point shifts to the right as the

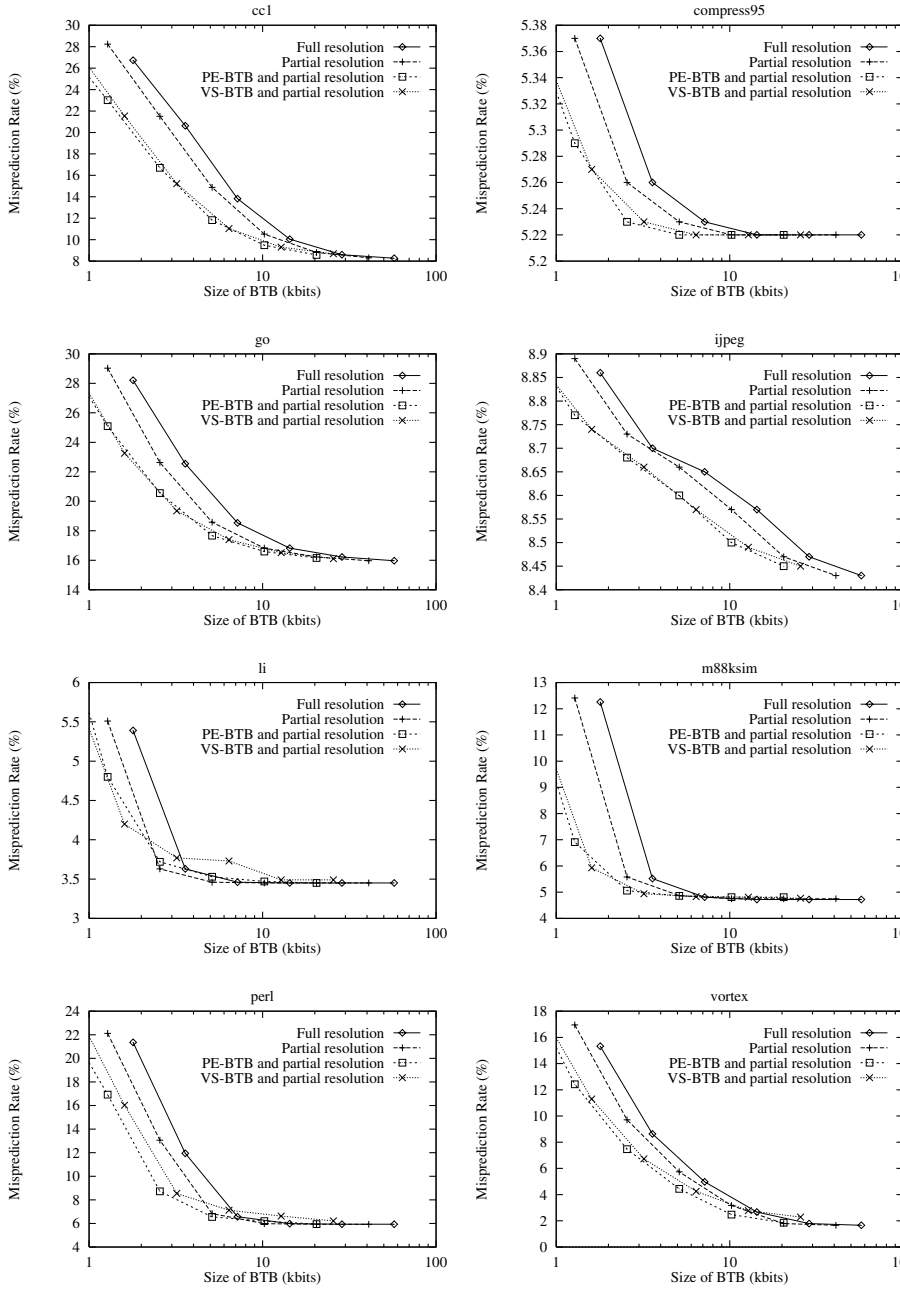


Fig. 4. BTB size vs. mispredict rate for the BTB types listed in Table 1 and shown in Fig. 1.

Type	Resolution	Target size	Type info size	Entry size	Fig. 1
Traditional	Full – 23-27b	4×30	2	220-236	(a)
Traditional	Partial – 8b	4×30	2	160	(b)
PE-BTB	Partial – 8b	$2 \times 30/4 \times 10/1 \times 30 + 2 \times 10$	2	82	(d)
VS-BTB	Partial – 8b	$4 \times 10/1 \times 30 + 3 \times 10$	2	100	(c)

Table 1. Four types of BTBs corresponding to the types shown in Fig. 1.

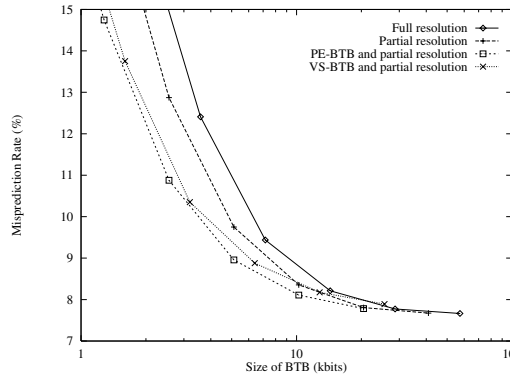


Fig. 5. BTB size vs. average mispredict rate for the BTB types listed in Table 1.

benchmarks become larger, and therefore more realistic. It will also shift to the right in a multi-tasking environment which puts more pressure on branch prediction resources.

The results show that both VS-BTBs and PE-BTBs have a better cost-efficiency than traditional BTBs, with PE-BTBs performing a little bit better than VS-BTBs. In the case of *li*, *perl*, and *vortex*, VS-BTB is performing worse than the other organisations for a large number of entries. The reason is conflicts between long branches that can only be mapped in one entry per set. This is clearly a disadvantage of VS-BTBs.

4.3 Variations

Several variations are possible on the VS-BTB and PE-BTB described above. First, one can use more than two target sizes. For example, 6 bits, 12 bits, and 30 bits. This could improve the utilisation of target fields.

PE-BTBs can be extended to three or more entries for long branches. This could be useful to match the total storage space of multiple short branches with the storage space of one long branch, especially when target size of short branches are very small, e.g., 6 bits, or the target size of a long branch is very large, e.g., 62 bits in a 64-bit architecture.

In the case of VS-BTBs, one can use several memories with a different number of entries for different branch target sizes. For example, one could use a table of 128 entries with 30 bit target fields together with a table of 512 entries with 10 bit target fields. The combination is a two-way set-associative BTB with the two ways having different numbers of entries.

5 Conclusions

Two methods for reducing the size and power dissipation of BTBs have been described: partial resolution and exploitation of branch locality. Partial resolution has been proposed before while exploiting branch locality is a contribution of this work. Partial resolution reduces the number of tag bits. This reduces the size of a BTB at the cost of a slightly higher misprediction rate due to false-hits. Two BTB organisations have been proposed for exploiting branch locality. In both organisations a distinction is made between short branches and long branches, with short branches being branches with which a certain number of the highest pc bits do not change when the branch is taken. In the VS-BTB organisation, one or more entries of a set (both proposed organisations assume set-associativity) can store both long and short branches while the others can store only short branches. In the PE-BTB organisation, entries can store only a short branch and two entries are required to store a long branch. These paired entries are adjacent and aligned for simplicity. The effectiveness of partial resolution, VS-BTBs, and PE-BTBs has been demonstrated by means of simulations. The results show a clear improvement in cost-efficiency for 32-bit architectures, with PE-BTBs being sightly more efficient than VS-BTBs. For 64-bit architectures, the improvement that can be realized with the described techniques will be even greater.

References

[1] Johnny K. F. Lee and Alan Jay Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Micro*, 21(7):6–22, January 1984.

[2] James E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–147, May 1981.

[3] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, October 12–15, 1992.

[4] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, Albuquerque, New Mexico, November 18–20, 1991.

[5] Scott McFarling. Combining Branch Predictors. Technical Report TN-36, Western Research Laboratory, Palo Alto, California, June 1993.

[6] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, Toronto, Ontario, May 27–30, 1991.

[7] Enric Musoll. Predicting the Usefulness of a Block Result: A Micro-Architectural Technique for High-Performance Low-Power Processors. In *Proceedings of the 32th Annual International Symposium on Microarchitecture*, pages 238–247, Haifa, Israel, November 16–18 1999.

[8] Barry Fagin and Kathryn Russell. Partial Resolution in Branch Target Buffers. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 193–198, Ann Arbor, Michigan, November 29–December 1, 1995.

[9] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, University of Wisconsin-Madison, Computer Sciences Department, June 1997.

Two-Level Address Storage and Address Prediction

Enric Morancho, José María Llabería and Àngel Olivé

Computer Architecture Department - Universitat Politècnica de Catalunya (Spain)[1]

Abstract. : The amount of information recorded in the prediction tables of the address predictors turns out to be comparable to current on-chip cache sizes. To reduce their area cost, we consider the spatial-locality property of memory references. We propose to split the addresses in two parts (high-order bits and low-order bits) and record them in different tables. This organization allows to record only once every unique high-order bits. We use it in a last-address predictor and our evaluations show that it produces significant area-cost reductions (28%-60%) without performance decreases.

1 Introduction

True-data and control dependencies are the major bottleneck for exploiting ILP. Some works [4][6][11] propose the use of prediction and speculation to overcome data dependencies. In load instructions, there is a true-data dependence between the address computation and the memory access; this dependence contributes to the large latency of the load instructions and can affect processor performance. Then, address predictors are valuable to access memory speculatively [4][10].

A typical address-prediction model is the last-address [11]. It assumes that a load instruction will compute the same address that the one computed in its previous execution. Proposals of last-address predictors [4][6] employ a direct-mapped Address Table (AT), indexed with some bits of the PC. Each AT entry contains the last address computed by a load instruction, a two-bit confidence counter, and a tag. We name this predictor *Base Last-Address Predictor* (BP).

Last address predictors use prediction tables that record up to 4.096 64-bit addresses [2][10], that is, 32 Kbytes; that is comparable to current on-chip cache sizes. However, the previous designs do not exploit the locality of the addresses. This property has been used in other works to take different advantages [3][12][13] (a detailed description of related works can be found in [9]). In this paper, we propose a new organization of the prediction table and we apply it to a typical last-address predictor to obtain a significant area-cost reduction.

This paper is organized as follows. Section 2 presents our proposal. Section 3 evaluates our proposal, and Section 4 summarizes the conclusions of this work.

---

1. *Author's address:* Computer Architecture Department, Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08034 Barcelona (Spain). *E-mail:* enricm@ac.upc.es

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 960-964, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Two-Level Address Predictor

2.1 Basic Idea

Effective addresses exhibit temporal and spatial locality; it produces redundancy in AT contents. For instance, different accesses to the same global variable produce temporal redundancy. Also, variables stored in consecutive addresses and stack accesses produce spatial redundancy. We propose an organization to record them non redundantly.

AT is split in two parts: the Low-Address Table (LAT) and the High-Address Table (HAT). LAT records the low-order bits of the addresses and HAT the high-order bits; moreover, each LAT entry is linked to a HAT entry. Then, a HAT entry can be shared by several LAT entries. We apply this organization to the BP and we obtain the *Two-Level Address Predictor* (2LAP); Figure 1 shows its scheme.

To predict a load instruction, 2LAP indexes LAT using the load-instruction PC; this access obtains the low-order address portion and a link to HAT. After that, HAT is accessed to obtain the high-order portion. This sequential access does not imply an implementation restriction because LAT can be accessed early in the pipeline and the large number of pipeline stages before issuing an instruction; for instance, 5 stages in an Alpha 21264 [1]. Moreover, we could reduce the critical path of the speculative access by recording in LAT enough bits for indexing the cache.

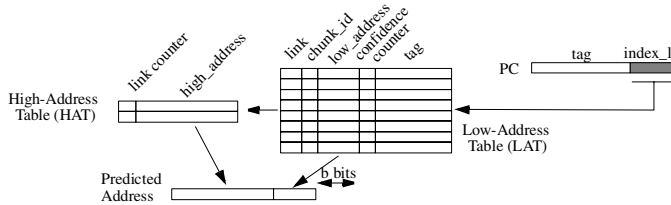


Fig. 1. Two-Level Address Predictor.

2.2 Locality Analysis and HAT Size

There is a trade-off between the number of HAT entries and the bits of the low-order portions (*b*). To obtain recommended sizes, we evaluate the locality in AT contents [9]; the suggestion is *b*=10, 12 or 14 and 64 HAT entries. We choose this HAT size because up to 96-entry HAT's can be accessed fully associatively in a single processor cycle [5].

2.3 Prediction-Table Management

2LAP updates LAT like BP updates AT, using the *always-allocate* policy. Also, before recording a high-order portion in HAT, 2LAP must verify if it is yet recorded. To check it, 2LAP looks for the high-order portion in HAT. If it is found (HAT hit), both entries are linked; if not, a HAT entry is evicted.

HAT Replacement and Tracing Empty HAT Entries

To reduce the eviction from HAT of useful information, we trace HAT entries not related to any LAT entry (empty HAT entries). To detect it, we relate to every HAT entry a link counter that reflects the number of LAT entries linked to it (*link_counter* in Figure 1). If none empty HAT entry is found, the replacement algorithm selects

randomly one HAT entry but the MRU one (no-MRU replacement algorithm). We have used this algorithm because the implementation of LRU algorithm is complex and expensive for large tables. We have evaluated the performance decrease produced by this decision: it is limited by 2.4% for b=10, 1% for b=12, and negligible for b=14.

2LAP updates the link counter on LAT replacements and on changes of the high-order portion related to a LAT entry. On HAT replacements, the LAT entries linked to the evicted HAT entry are not invalidated to simplify the design. This decision is possible because 2LAP is related to an speculative mechanism, but it produces mispredictions. Our experiments show that using three-bit link counters the performance of the 2LAP is almost saturated. These counters estimate empty HAT entries with a high correctness.

Filtering-Out Some High-Order Portions in HAT

2LAP allocates unpredictable load instructions in LAT but avoids the allocation in HAT of their high-order portions, i.e., their LAT entries are not linked to any HAT entry. Moreover, the link is broken when the classification of a load instruction changes from predictable to unpredictable, and only is re-established when it changes again.

The address chunk stored in *low_address* field of LAT is used to keep on updating the classification of the unpredictable instructions; the basic idea of this classification has been proposed in [7]. Also, the chunk is selected dynamically (*chunk_id* field in Figure 1) because the accuracy of 2LAP in programs with large-strided references (*ijpeg*) can be affected.

Filtering HAT Allocations and Managing Empty HAT Entries

We have evaluated the four possibilities that appear considering: a) filtering-out HAT allocations, and b) managing empty HAT entries. Our experiments show that both policies should be applied at the same time, specially in codes with a large working set of high order portions and medium-predictable instructions (*gcc*, *go*, *vortex*).

3 Evaluation: 2LAP versus BP

This section compares 2LAP versus BP, both using bounded prediction tables. Working-set size of static load instructions of the programs [8] justifies that the selected LAT sizes range from 256 to 4.096 entries.

3.1 Area Cost of the Predictors

We evaluate the area cost of a predictor as the amount of information that it records. Following formulas show the area cost of BP and 2LAP using 64-bit logical addresses, t-bit tags, 3-bit link counters, and b-bit address chunks.

$$\underset{AreaCost}{BasePredictor} = (t + 64 + 2) \times ATentries$$

$$\underset{AreaCost}{2LAP} = (3 + (64 - b)) \times HATentries + \left(\log_2 HATentries + \left\lceil \log_2 \frac{64}{b} \right\rceil + b + 2 + t\right) \times LATentries + \log_2 HATentries$$

Tag length influences on predictor accuracy. In the analysed codes, [8] shows that BP accuracy saturates when the number of index and tag bits is 17; then, we compare these configurations. The area-cost reduction from a BP to a 2LAP with the same number of AT and LAT entries ranges from 37% (256 entries, b=14) to 60% (4.096 entries, b=10).

3.2 Captured Address Predictability

The captured address predictability is defined as the percentage of correct predictions out of the number of executed load instructions. We will evaluate the predictability captured by several predictor configurations in the integer codes of the SPEC-95 with the largest working-set size of static load instructions (remaining programs present a similar behaviour but in a different table-size range). Our results were obtained running Alpha binaries instrumented with ATOM; programs were run until completion using reference input sets.

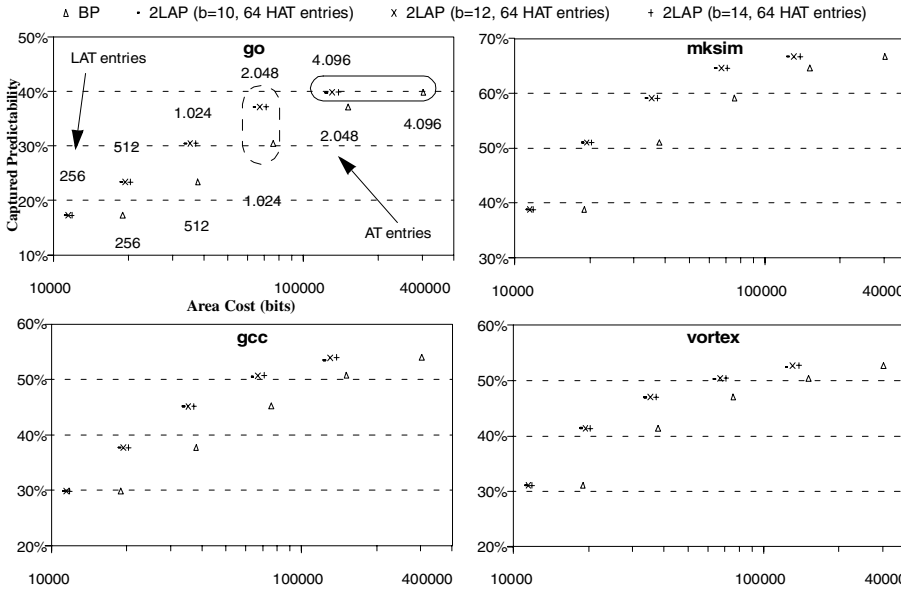


Fig. 2. Predictability captured by BP and 2LAP in several benchmarks. Horizontal axes stand for base-10 logarithm of predictor area cost, vertical axes stand for captured predictability.

Figure 2 shows the predictability captured by 2LAP and BP. Horizontal axes stand for area cost and vertical axes for captured predictability. Leftmost top graph is labelled with the number of AT and LAT entries of the configurations.Area-cost reduction from a 2LAP to a BP with the same number of LAT and AT entries do not represent a performance loss; for AT entries=LAT entries=4.096, a continuous oval surrounds these configurations. When LAT entries=2×AT entries (configurations surrounded by a dashed oval for LAT entries=2×AT entries=2.048), we obtain configurations with similar area cost. 2LAP outperform BP because LAT has less capacity misses than AT.

3.3 Accuracy

The accuracy of a predictor is defined as the percent of correct predictions out of the number of predictions. As every misprediction could produce a penalty of several processor cycles, the 2LAP should not present a lower accuracy than the BP. Our evaluations show that for b=10, 2LAP presents a slightly lower accuracy than the BP (in the worst benchmark -*gcc*- the difference is limited by 0.7%). For b=14, the difference is negligible. We present in [9] a detailed accuracy comparison.

4 Conclusions

We have shown that the spatial-locality property of the memory references produces redundancy in the prediction tables. We have taken advantage of this fact to reduce the area cost of the prediction tables. Our proposal splits the addresses computed by the load instructions in two parts: high-order and low-order portion. Addresses with the same high-order portion are recorded sharing one copy of the portion. Also, management of empty HAT entries, and filtering-out allocations of high-order portions related to unpredictable instructions improve the performance of the proposal.

Other prediction models (stride, context and hybrid) can also take advantage of the locality of the addresses to reduce their area cost.

This work proposes a new organization of the prediction table but it maintains the allocation policy, then, our proposal predicts the same instructions than the BP, and IPC speed-up is the one reported in other works [2][4][10].

Acknowledgements

This work was supported by the spanish government (grant CICYT TIC98 0511-C02-01), and the CEPBA (European Centre for Parallelism of Barcelona)

References

1. Alpha 21264 MicroProcessor Data Sheet. (1999). Compaq Computer Corporation.
2. B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai and J.P. Shen. (1998). Load Execution Latency Reduction. In *ICS-12*, pp. 29-36
3. M. Farrens and A. Park. (1991). Dynamic Base Register Caching: A Technique for Reducing Address Bus Width. In *ISCA-18*, pp. 128-137.
4. J. González and A. González. (1997). Speculative Execution via Address Prediction and Data Prefetching. In *ICS-11*.
5. B. Jacob and T. Mudge. (1998). Virtual Memory in Contemporary Microprocessors. IEEE Micro, Vol 18(4), pp. 60-75.
6. M.H. Lipasti, C. B. Wilkerson and J.P. Shen. (1996). Value Locality and Load Value Prediction. In *ASPLOS-7*.
7. E. Morancho, J.M. Llabería and À. Olivé. (1998). Split Last Address Predictor. In *PACT'98*.
8. E. Morancho, J.M. Llabería and À. Olivé. (1999). *Looking at History to Filter Allocations in Prediction Tables*. In *PACT99*, pp. 314-319
9. E.Morancho, J.M. Llabería and À. Olivé. (1999). Two Level Address Storage and Address Prediction. Technical report UPC-DAC-99/48.
10. G. Reinman and B. Calder. (1998). Predictive Techniques for Aggresive Load Speculation. In *MICRO-31*.
11. Y. Sazeides and J.E. Smith. (1996). The Predictability of Data Values. In *MICRO-29*.
12. A. Seznec. (1994). Decoupled sectored caches: reconciliating low tag volume and low miss rate. In *ISCA-21*.
13. H. Wang, T. Sung and Q. Yang. (1997). Minimizing Area Cost of On-Chip Cache Memories by Caching Address Tags. *IEEE Transactions on Computers*, 46 (11).

Hashed Addressed Caches for Embedded Pointer Based Codes

Marian Stanca, Stamatis Vassiliadis, Sorin Cotofana, and Henk Corporaal

Electrical Engineering Department
Delft University of Technology
Delft, The Netherlands
{akela, stamatis, sorin, heco}@cardit.et.tudelft.nl

Abstract. We are proposing a cache addressing scheme based on hashing intended to decrease the miss ratio of small size caches. The main intention is to improve the hit ratio for 'random' patterns pointer memory accesses for embedded (special purpose) system applications. We introduce a hashing scheme, denoted as bit juggling, and measure the effect such a scheme has in the cache access miss ratio. It is shown, for the considered benchmark, that 3-bit bit juggling will reduce the miss ratio for up to 12%, for associative caches of maximum size of 16KBytes when compared to usual cache addressing schemes.

1 Introduction

In this paper we address the issue of lowering the cache miss ratio for pointer based codes. We assume that we have pointer based applications that are meant to be executed on an architecture that includes a cache memory. Moreover we consider that we are at liberty to change the hardware implementation of that cache memory. More in particular we address the issue of hashing the cache memory for pointer based codes.

To evaluate the effectiveness of hashed caches we propose a framework which is based on a simulation tool. Our framework includes a modified cache simulation tool, based on the Simple Scalar[1]. Our experiments suggested the following:

- Hashed caches can lower the cache miss ratio for certain types of applications, while worsening the performance for others.
- The effectiveness of the approach can be assessed only with a priori knowledge of the memory access patterns.
- The implementation cost for the necessary extra hardware is negligeable (mostly inexistent).
- Improvement in miss ratio after applying the hashing scheme is up to 12%.

2 Hashing Functions and Bit Juggling Addressing

An address issued by the CPU has three different components: a tag used to identify if a line is in the cache memory, a line index to choose one of m entries

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 965–968, 2000.
© Springer-Verlag Berlin Heidelberg 2000

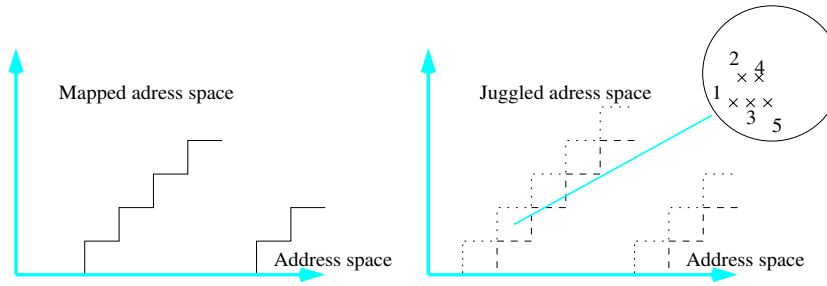


Fig. 1. Normal and hashed address mapping

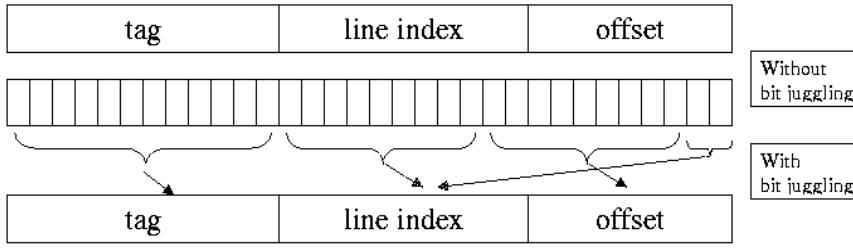


Fig. 2. Tag, line index and offset extracting from issued address

from the m-way set associative cache, and a line offset to choose one memory location from the cache line. The usual addressing of direct mapped caches and set associative caches are depicted in Equation 1 and Equation 2 respectively.

$$F_{hashing} = \text{(Block address) MOD (\# of blocks in the cache memory)} \quad (1)$$

$$F_{hashing} = \text{(Block address) MOD (\# of sets in the cache memory)} \quad (2)$$

Our approach is using hashing functions in order to minimize the traffic between two consecutive memory levels. The usual hashing function (MOD) used to map main memory addresses into cache memory addresses can be seen in Figure 1, as well as a different hashing function. Certain bits used for line index or offset are extracted from the address issued by the CPU as seen in Figure 2. We are introducing a hashing technique for caches, named bit juggling (BJ). Instead of using the number of necessary bits starting with position 0 to the left for line offset we are using successive bits starting from at least position 1. The necessary tag bits will be the same. The line index bits will be the remaining ones. If, for example, we are using 1-bit BJ, the memory accesses patterns will change as follows: odd address memory accesses in the range of two line size will be stored consecutively in the same line. The line placement mechanism will remain the same except for the notion of successiveness. Successive addresses will now be, in the cache memory, those separated by $2^{bitjuggled}$ addresses. Accommodating

Table 1. Data cache miss rates for various configurations

Line size →	8	8	16	16	32	32
Cache size	-	BJ3	-	BJ3	-	BJ3
512Bytes(2)	**45.10**	**36.00**	**43.65**	**35.26**	**41.43**	**34.82**
4KBytes(2)	**39.11**	**28.76**	**37.15**	**27.93**	**36.72**	**25.12**
8KBytes(2)	**34.38**	**24.77**	**33.87**	**22.15**	**33.15**	**20.10**
16KBytes(2)	**28.25**	**19.12**	**27.96**	**18.02**	**27.03**	**16.53**
512Bytes(4)	**43.44**	**33.17**	**40.34**	**32.23**	**38.52**	**29.48**
4KBytes(4)	**37.12**	**27.19**	**35.40**	**26.20**	**31.21**	**23.17**
8KBytes(4)	**29.29**	**18.31**	**26.14**	**16.26**	**22.46**	**12.32**
16KBytes(4)	**24.37**	**13.28**	**21.55**	**11.41**	**17.67**	**8.60**

such a technique will necessitate changes to the cache buffers only. In order to simplify the approach we will assume that we may choose the main memory organization's in banks. The number of banks is specified by the number of bits juggled. Therefore, next memory level organization's will allow reading and writing with strides. The line identification process needs the extension of the tag, with a number of redundant bits. The extra number of bits are the number of bits to be juggled. The line replacement mechanism remains the same, according with the cache memory associativity.

3 Evaluation

We used the Simple Scalar simulation tool[1]. We have modified the cache simulator in order to accommodate our technique. The modifications performed were pertaining the addressing mode of the cache, the hit/miss handler routine, and the replacement algorithm. 'Pointer based' and 'big enough to observe relevant statistical phenomenons' were two key characteristics when choosing benchmarks to test our technique. The pointer based C implementation of MPEG2[2] is the only benchmark with the right characteristics. After obtaining an instrumented executable, namely the MPEG2 encoder / decoder, we simulated it with our modified tool. The results presented are for the MPEG2 decoder, the encoder shows similar results and performance improvements patterns. We simulated an MPEG2 decoder with a 476 frames mpeg stream, with 300 x 200 pixels image size. The results are presented in Table 1 for a number of cache sizes, cache associativities of 2 and 4, and for various cache line sizes. Due to implementation particularities cache miss ratio varies proportionally with the frame stream size(weak) and frame size(strong). The data stream is organized in 8 x 8Bytes objects(searched and identified in successive frames), or smaller. Due to this reason we had to keep the line size smaller than the customary 32-128Bytes in general computing in order to observe relevant patterns in memory accesses in various parts of the application. Because of the small cache line size the number of first reference cache misses was increased. The simulations were performed

Table 2. Data cache miss rates for 64Bytes cache size, 8Bytes line size

	4 entries, associativity 2	2 entries, associativity 4
No Bit Juggling	34.91	32.34
Bit Juggling 3bits	25.17	21.75

without a second level cache memory. Due to the data stream organization the 3-bits BJ was our prime candidate to test our technique. The improvement in performance was up to 12%. For the direct mapped cache, a cache line can be stored in one place only and collisions are solved by replacing the old cache line with the new one. Because of this fact a change in the memory access patterns for caches with associativity of 1 will not show notable modifications in performance. Therefore we concentrate our efforts on cache memories with greater associativity. Caches smaller then 4KBytes are not likely to be found in general purpose computing but can be found in embedded applications. In our simulations we assumed small cache sizes intended for application specific inexpensive engines. For comparison reasons we have included some simulation figures for a very small cache (64Bytes) in Table 2, for a line size of 8Bytes and a MPEG stream containing only 34 frames, with 150 x 100 pixels image size. We can conclude that the MPEG2 pointer based implementation should be ran on an architecture with a hashed cache with 3 bits bit juggling technique.

Array or record based codes with accesses in strides may be also used to test this technique. In this case deterministic determination of the number of bits to be juggled is possible from inspection.

4 Conclusions and Future Work

A new technique to decrease the cache miss ratio, denoted bit juggling, for pointer based codes has been introduced. Bit juggling has the characteristic of any hashing process, namely is offering improvement or worsening of performance according to the application. For a sample of pointer based code it has been shown that three bits bit juggling is offering good relative performance improvements. Using the same technique with lesser bits shows degradation in performance. The success in applicability of this technique is related with the amount of information on memory access patterns a priori known. Embedded applications are, therefore, a primary target.

References

1. D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
2. M. S. S. Group. Mpeg-2 video codec, http://www.mpeg.org, 1996.
3. G. D. Knott. Hashing functions. *The Computer Journal*, 18(3):265–278, Aug. 1975.

BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations*

Mihai Budiu[1], Majd Sakr[2], Kip Walker[1], and Seth C. Goldstein[1]

[1] Carnegie Mellon University
{mihaib,kwalker,seth}@cs.cmu.edu
[2] Pittsburgh University
sakr@ee.pitt.edu

Abstract. We present a compiler algorithm called BitValue, which can discover both unused and constant bits in dusty-deck C programs. BitValue uses forward and backward dataflow analyses, generalizing constant-folding and dead-code detection at the bit-level. This algorithm enables compiler optimizations which target special processor architectures for computing on non-standard bitwidths.
Using this algorithm we show that up to 31% of the computed bytes are thrown away (for programs from SpecINT95 and Mediabench). A compiler for reconfigurable hardware uses this algorithm to achieve substantial reductions (up to 20-fold) in the size of the synthesized circuits.

1 Introduction

As the natural word width of processors increases, so grows the gap between the number of bits used and those actually required for a computation. Recent architectural proposals have addressed this inefficiency by providing collections of narrow functional units or the ability to construct functional units on the fly. For example, instruction set extensions which support subword parallelism (e.g., [10]), Application-Specific Instruction-set Processors (ASIPs) (e.g., [9]), and reconfigurable devices (e.g., [11]) all allow operations to be performed on operands which are smaller than the natural word size.

Reconfigurable computing devices are the most efficient at supporting arbitrary size operands because they can be programmed post-fabrication to implement functions directly as hardware circuits. In such devices, functional units are created which exactly match the bit-widths of the data values on which they compute.

Using the special architectural features requires the programmer to use macro libraries or specify the bit-widths manually, a tedious and error-prone process. Furthermore, this is often impossible as there is little or no support in high-level languages for specifying arbitrary bit-widths.

* This work was supported by DARPA contract DABT63-96-C-0083 and an NSF CAREER grant.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 969–979, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In this paper we present the BitValue algorithm, which enables the compilation of unannotated high-level languages to take advantage of variable size functional units. Our technique uses dataflow analysis to discover bits which are independent of the inputs to the program (constant bits) and bits which do not influence the output of the program (unused bits). By eliminating computation of both constant and unused bits the resulting program can be made more efficient.

BitValue generalizes constant folding and dead-code elimination to operate on individual bits. When used on C programs, BitValue determines that a significant number of the bit operations performed are unnecessary: on average 14% of the computed bytes in programs from SpecINT95 and Mediabench are useless. Our technique also enables the programmer to use standard language constructs to pass width information to the compiler using masking operations.

Narrow width information can be used to help create code for sub-word parallel functional units. It can also be used to automatically find configurations for reconfigurable devices. BitValue has been implemented in a compiler which generates configurations for reconfigurable devices, reducing circuit size by factors of three to twenty.

In Section 2 we present our BitValue inference algorithm with an example. Results for the implementation in a C compiler are in Section 3 and for a reconfigurable hardware compiler in Section 4. Related work is presented in Section 5 and we conclude in Section 6.

2 The BitValue Inference Algorithm

For each bit of an arbitrary-precision integer, our algorithm determines whether (1) it has a constant value, or (2) its value does not influence the visible outputs of the program. Those two possibilities are similar to constant folding and dead code elimination, respectively. In our setting, however, these are performed at the bit-level within each word.

We can cast our problem as a *type-inference* problem, where the type of a variable describes the possible value that each bit can have during the execution of the program. The BitValue algorithm solves this problem using dataflow analysis.

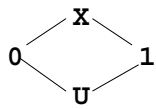


Fig. 1. *The bit values lattice.*

We represent the bit values by one of: $\langle \mathtt{0} \rangle$, $\langle \mathtt{1} \rangle$, *don't know* (denoted by $\langle \mathtt{u} \rangle$) and *don't care*, (denoted by $\langle \mathtt{x} \rangle$). Let us call this set of values $\mathcal{B}$. Some bits are constant, independent of the inputs and control flow of the program; such bits are labeled with their value, $\langle \mathtt{0} \rangle$ or $\langle \mathtt{1} \rangle$. A bit is labeled $\langle \mathtt{x} \rangle$ if it does not affect the output; otherwise a bit is labeled $\langle \mathtt{u} \rangle$. These bit values form a lattice, depicted in Figure 1. We write $\cup$ and $\cap$ for sup and inf in the lattice respectively. The top element of the lattice is $\langle \mathtt{x} \rangle$ and the bottom is $\langle \mathtt{u} \rangle$.

The Bit String Lattice. We represent the type of each value in the program as a string of bits. We write $\mathcal{B}^*$ to denote all strings of values in $\mathcal{B}$. For example, for the C statement `unsigned char a = b & 0xf0`, we determine that the type of `a` is $\langle$`uuuu0000`$\rangle$, and that the type of `b`, assuming it is dead after this statement, is $\langle$`uuuuxxxx`$\rangle$.

The bitstrings also form a lattice $\mathcal{L}$. Space considerations preclude us from giving the formal definition of the operations on this lattice. The $\cup$ and $\cap$ operations in $\mathcal{L}$ are done bitwise (i.e. $ab \cup cd = (a \cup c)(b \cup d)$). When applied to strings of different lengths, $\cup$ gives a result of the shorter length, while $\cap$ gives a result of the bigger length. The shorter value is sign-extended in the lattice for the $\cap$ computation.

The Transfer Functions. In order to certify the correctness of our algorithms, we need to prove that our transfer functions are monotone and conservative. For this purpose we provide mathematical definitions for the "best" forward transfer function and for a conservative backward transfer function.

We now define $\mathcal{A}$, the forward transfer function of an operator in $\mathcal{L}$. We define the auxiliary "expansion" function $\exp : \mathcal{L} \times \{\langle \mathtt{x} \rangle, \langle \mathtt{u} \rangle\} \to 2^{\mathcal{L}}$, which takes a bitstring s and a bit value $b \in \{\langle \mathtt{x} \rangle, \langle \mathtt{u} \rangle\}$, and generates a set of bistrings: all bitstrings that can be obtained from s by replacing the bits in s having the value b by all possible combinations of constant values. For example $\exp(\langle \mathtt{0ux1x} \rangle, \langle \mathtt{x} \rangle) = \{\langle \mathtt{0u010} \rangle, \langle \mathtt{0u110} \rangle, \langle \mathtt{0u011} \rangle, \langle \mathtt{0u111} \rangle\}$.

We now define three auxiliary functions which are used to compute the transfer function of any operator. $\mathcal{A}_c : (\mathbb{N} \to \mathbb{N}) \times \{0,1\}^* \to \mathcal{L}$ operates on "constant" bitstrings, i.e. bistrings containing only $\langle \mathtt{0} \rangle$ and $\langle \mathtt{1} \rangle$. $\mathcal{A}_u : (\mathbb{N} \to \mathbb{N}) \times \{\langle \mathtt{0} \rangle, \langle \mathtt{1} \rangle, \langle \mathtt{u} \rangle\} \to \mathcal{L}$ computes the transfer function for bitstrings which comprise $\langle \mathtt{0} \rangle$, $\langle \mathtt{1} \rangle$ and $\langle \mathtt{u} \rangle$ bits. $\mathcal{A} : (\mathbb{N} \to \mathbb{N}) \times \mathcal{L} \to \mathcal{L}$ works for bitstrings with any of the digits in $\mathcal{B}$. Given a unary operation $f : \mathbb{N} \to \mathbb{N}$, $\mathcal{A}(f, \cdot)$ is its associated forward transfer function in $\mathcal{L} \to \mathcal{L}$.

$$
\begin{array}{lll}
\mathcal{A}_c(f,v) & = f(\text{value}(v)) & \text{where } v \in \{\langle \mathtt{0} \rangle, \langle \mathtt{1} \rangle\}^* \\
\mathcal{A}_u(f,v) & = \bigcap_{y \in \exp(v, \langle \mathtt{u} \rangle)} \mathcal{A}_c(f,y) & \text{where } v \in \{\langle \mathtt{0} \rangle, \langle \mathtt{1} \rangle, \langle \mathtt{u} \rangle\}^* \\
\mathcal{A}(f,v) & = \bigcup_{y \in \exp(v, \langle \mathtt{x} \rangle)} \mathcal{A}_u(f,y) & \text{where } v \in \mathcal{L}.
\end{array}
$$

The intuition behind these equations is the following: when we compute the transfer function in $\mathcal{L}$ for an input value, we can choose arbitrary values for the input bits which are marked $\langle \mathtt{x} \rangle$, but we must search the entire space of possibilities for the bits marked $\langle \mathtt{u} \rangle$. This definition can be easily extended to deal with n-ary operators.

For example, here is what the above definition yields for the C complementation ~ operator when applied to $\langle$`u0x`$\rangle$:

$$\begin{aligned}
\mathcal{A}(\tilde{}, \langle \texttt{u0x} \rangle) &= \mathcal{A}_u(\tilde{}, \langle \texttt{u00} \rangle) \cup \mathcal{A}_u(\tilde{}, \langle \texttt{u01} \rangle) \\
&= (\mathcal{A}_c(\tilde{}, \langle \texttt{000} \rangle) \cap \mathcal{A}_c(\tilde{}, \langle \texttt{100} \rangle)) \cup (\mathcal{A}_c(\tilde{}, \langle \texttt{001} \rangle) \cap \mathcal{A}_c(\tilde{}, \langle \texttt{101} \rangle)) \\
&= ((\tilde{}\langle \texttt{000} \rangle \cap \tilde{}\langle \texttt{100} \rangle) \cup (\tilde{}\langle \texttt{001} \rangle \cap \tilde{}\langle \texttt{101} \rangle)) \\
&= ((\langle \texttt{111} \rangle \cap \langle \texttt{011} \rangle) \cup (\langle \texttt{110} \rangle \cap \langle \texttt{010} \rangle)) \\
&= \langle \texttt{u11} \rangle \cup \langle \texttt{u10} \rangle \\
&= \langle \texttt{u1x} \rangle
\end{aligned}$$

The backward transfer function will discover *don't care* bits in the input starting from the *don't care*s in the output. We do not have a closed form for the backward transfer function. We can, however, define a conservative approximation using techniques from Boolean function minimization [6]. The notion of *don't care* input for a Boolean function f of n variables is well known (x_i is a *don't care* if $\frac{\partial f}{\partial x_i} = 0$).

We can view an operator which computes many bits (like addition) as a vector of Boolean functions, each computing one bit of the result. An input bit is don't care for the operator if it is a don't care for *all* the functions in the vector whose result is not $\langle \texttt{x} \rangle$. If our analysis discovers that some input bits are constant, we can use those in the backward transfer function computation, starting the computation with the restriction of f to those constant inputs.

For example, let us see how the backward propagation operates on the statement `c = a^b` when we know already that the types of `a`, `b` and `c` are respectively $\langle \texttt{u0} \rangle$, $\langle \texttt{uu} \rangle$ and $\langle \texttt{xu} \rangle$; we expect the don't care of `c` to be propagated to `a` and `b`. The two bits of `c` are computed by two boolean functions of 4 bits: f_0 and f_1: $f_0(a_0 = 0, a_1, b_0, b_1) = a_0 \hat{} b_0$ and $f_1(a_0, a_1, b_0, b_1) = a_1 \hat{} b_1$. Because the bit 1 in the result is $\langle \texttt{x} \rangle$, we only need to look for the *don't care*s of f_0 which will be the *don't care*s of the input. For instance, bit a_1 is a *don't care*, because $f_0|_{a_1=0} = f_0|_{a_1=1}$, and so is b_1. So the backward propagation proves as expected that the types of the inputs are $\langle \texttt{x0} \rangle$ and $\langle \texttt{xu} \rangle$ respectively. In this example the fact that $a_0 = 0$ was not useful to infer more information, but if we change the operator from `^` to `&`, this information provides the type $\langle \texttt{x} \rangle$ for b_0.

In practice the transfer functions as given by the above definitions can be expensive to compute, so we resort to using monotone conservative approximations, described fully in [5].

The Dataflow Analysis. We maintain for each value two types: the *best* type and the *current* type. The *best* type is initialized conservatively ($\perp$) and moves up in the lattice after each pass. The analysis works by alternating forward and backward dataflow passes, terminating when the *best* type does not change during a pass. Each pass starts by initializing the *current* type for all the values to $\top$, and proceeds to do the dataflow computation; during this computation the *current* types move down in the lattice until a fixed point is reached. At the end of each pass we update the *best* type: $best = best \cup current$.

```
unsigned char
f(unsigned char c,
  unsigned char a)
{
    unsigned char d;
    d = (c + a) & 0x33;
    return (d >> 4)
           + (d << 2);
}
```

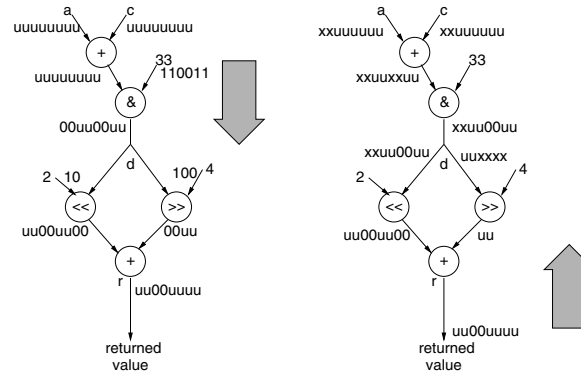


Fig. 2. *A C function and the associated data-flow graph. The types inferred by forward (backward) propagation are shown in the left (righ) figure. We assume that a* `char` *has 8 bits.*

2.1 Example

We illustrate the algorithm on the code in Figure 2.[1] The algorithm begins with the forward pass and examines the first statement. The sources for the first statement are parameters which are defined outside the procedure and thus are set to be all *don't knows*, i.e. every bit is significant. `c+a` from Figure 2 must be computed on 9 bits. The result is truncated to 8 bits of precision because of the definition of `char` in the underlying language implementation. The masking operation creates a type for `d` with a combination of constants and *don't knows*, ⟨`00uu00uu`⟩.

The left shift in the return statement concatenates 0 bits at the least significant end, while the right shift generates the type ⟨`00uu`⟩. Using this information, the addition in the `return` statement infers that the final result has type ⟨`uu00uuuu`⟩.

The backward pass uses this information as a starting point. It proceeds to determine which bits of the computation are actually needed. In this example, the right shift indicates that the bottom 4 bits of `d` are *don't cares*, and the left shift indicates that the top 2 bits are *don't cares*. Since `d` is used in two expressions, its useful bits are represented by the ∩ of these two strings. The middle two bits of `d` have been found to be 0 by forward propagation, and they are not changed.

From the `&` we deduce that the useful bits of the sum `a+c` are ⟨`xxuuxxuu`⟩. This *don't care* information propagates up through the transfer function associated with the *plus* operation, and the compiler deduces that for both `a` and `c` only the bottom 6 bits are significant.

During the next forward pass there are no changes and the algorithm terminates.

[1] We assume that all computations are carried on 8 bits; a normal C implementation would cast all values to `int` and back.

Table 1. *Percent reduction in bitwidth for programs in MediaBench (left) and SpecINT95 (right). We are only counting the most significant bytes. The column labeled "bitv" indicates that only the BitValue was run, "ind" indicates that only loop induction-variable analysis was performed, and "both" indicates both analyses were performed. The results are rounded down; a zero means "less than one percent". We were unable to profile gcc.*

	Static %			Dynamic %		
Benchmark	ind	bitv	both	ind	bitv	both
adpcm_e	0	19	19	0	19	19
adpcm_d	0	19	19	0	24	24
g721_Q_d	1	32	33	4	26	31
g721_Q_e	1	32	33	4	25	29
gsm_e	1	30	31	7	7	14
gsm_d	1	30	31	4	24	24
epic_e	0	5	5	0	0	0
epic_d	0	3	3	0	4	4
mpeg2_e	0	12	13	24	4	28
mpeg2_d	0	9	10	1	7	8
jpeg_e	0	4	4	1	7	9
jpeg_d	0	4	4	0	11	11
pegwit_e	0	14	15	0	13	13
pegwit_d	0	14	15	0	16	16
mesa	0	5	5	0	5	5

	Static %			Dynamic %		
Benchmark	ind	bitv	both	ind	bitv	both
124.m88ksim	1	22	22	1	19	20
129.compres	2	11	13	0	11	12
099.go	0	6	7	0	2	2
130.li	0	14	14	0	12	12
132.ijpeg	0	5	5	1	10	11
134.perl	0	11	11	0	8	8
147.vortex	0	6	6	0	5	5
126.gcc	0	19	19	*	*	*

3 Experiments with a C Compiler

We evaluate our algorithm implemented in SUIF [15] on C programs from MediaBench [8] and SpecINT95 [12]. BitValue is implemented as a work-list based dataflow algorithm starting from def-use chains [14]. Both def-use and BitValue are local analyses. Information from alias analysis or an interprocedural BitValue analysis would improve our results, at a cost of greater compilation time.

3.1 Evaluation

In this section we compare the merits of induction variable analysis, BitValue, and the interaction between them. Induction variable analysis has been used in [13] to compute ranges of values for each variable which is used to reduce the number of necessary bits. We have used a simplified form of this analysis to analyze FORTRAN-style `for` loops. We only detect values which depend linearly on the loop index.

We ran three experiments for each benchmark: the induction-variable analysis only, BitValue only, and both. When we ran both analyses, we first ran the induction-variable analysis, and we fed the bounds derived by it into the initial information for BitValue.

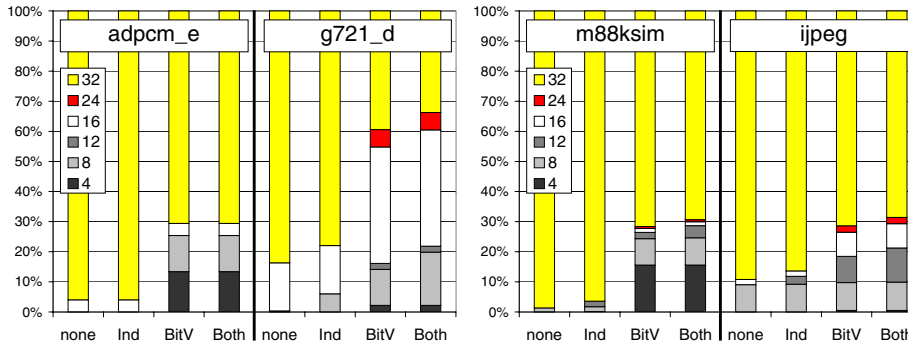


Fig. 3. *Percentage breakdown of widths from some programs (dynamic counts).*

Depending on the hardware model which exploits the narrow bitwidths, not every constant or *don't care* bit can be eliminated. For instance, a constant bit in the middle of a byte cannot be discarded when using subword parallelism. To account for this, the data in Table 1 counts only the most significant bytes as useless. For example, in a 16-bit data item with inferred type ⟨`x001u001 xxxuuuxx`⟩ we count *no* saved bytes because there is a useful bit in each of the bytes. These results underestimate the performance of the algorithm but apply to a wider range of architectures. If we count all the bit savings, we obtain on average an additional 6% reduction.

Most often BitValue and the induction-variable analysis complement (or even reinforce) each other: there are benchmarks (e.g., jpeg_e) where the "both" count surpasses the sum of the two other counts.

In Figure 3 we show the histograms of the data sizes operated upon for some selected benchmarks. The value sizes are rounded up. For each program we present four histograms: one for the original program (with no analysis), one for the induction variable analysis alone, one for BitValue analysis alone, and one for both analyses (induction followed by BitValue).

For example, we can interpret the graphs for `adpcm_e` in the following way: the first bar says that about 5% of the values in the original program are 16-bit or less. The fourth bar shows that using BitValue we discover that 16 bits are actually enough for about 30% of the values in the program.

We have examined the main sources of reductions to gain insight into the effectiveness of the algorithm. The sources of reduction found by BitValue come from several patterns: (1) the use of shift, bitwise and and or, addition and multiplication by small constants are the most powerful; (2) the propagation of cast information through the backward analysis; (3) array element index computations.

A preliminary evaluation of the benefits of discovering narrow values shows that the analysis is important in the context of reconfigurable functional units (RFUs). We used as a target architecture a VLIW processor augmented with an PipeRench-like [7] RFU on the data path. For example, compiling g721_e for the

processor+RFU combination without the analysis yields an 18% reduction in the running time. If we optimize the portion mapped to the RFU using BitValue we obtain a 26% reduction in running time.

3.2 Practical Issues

Our implementation of BitValue is fast and scales linearly in practice with program size. The space complexity is linear. We analyze on average 900 lines/second on a PIII @750Mhz, with an untuned implementation.

An interesting side-effect of our analysis is that it gives a portable high-level method for specifing widths: by using a masking operation we can seed the BitValue algorithm. For example, the statement `c = c & 0x3c` indicates that only the middle 4 four bits of `c` are useful, and this knowledge is propagated by BitValue throughout the code.

Our current implementation runs the induction variable analysis only once and BitValue afterwards. Improvements can be obtained by iterating these analyses until a fixed-point is reached. Future work will investigate the possible gains.

4 Experiments with a Reconfigurable Hardware Compiler

In this section we evaluate the BitValue algorithm as it is used in the DIL compiler [4] which we developed for reconfigurable hardware. The DIL language operates on arbitrary-precision integer data types and does not require the values to be annotated with an explicit width.

Because of this there is no baseline for comparing the performance of the algorithm (in C we could compare the reduced sizes with the C type-specified sizes). For evaluation purposes we artificially set the sizes of all variables to 32-bitsand then we run the algorithm to determine the reduction in size.

Table 2 shows the amount of hardware required to implement kernels compiled with the DIL compiler. Note that the impact of the analysis is significant: it can decrease the silicon real-estate (and implicitly, decrease the power consumption and decrease the latency of the computation) with a factor between 3 and 20.

5 Related Work

There is a wealth of static and dynamic analyses which suggest that many of the bits computed by a program are useless.

Brooks and Martonosi [3] use a simulator to show that for the programs in both SpecInt95 and MediaBench more than half of all integer computations require at most 16 bits of precision. Our compile time analysis proves statically that on average 30% of the widths are 16 bits or less for any input data. They suggest hardware techniques for creating instructions which operate on narrow widths on the fly. The work of Bondalapati and Prasanna is similar, looking

Table 2. *The size of the circuits in bit-operations/8, for two circuit versions: one where all values are 32-bit and one with variable sizes. The percent column shows the remaining size of the circuit after optimizations (the smaller, the better).*

Program	Description	Original	Final	%
cordic	12 stage implementation of Cordic vector rotations	1507	332	23
encoder	8-bit Huffman encoder with the code table hardwired	2286	578	26
dct	1-D 8-point Discrete Cosine Transform	366	94	26
fir	FIR filter with 20 taps and 8-bit coefficients	320	123	39
idea	Complete 8 round key-specific International Data Encryption Algorithm	2074	576	28
nqueens	Evaluator for the n-queens problem on an 8x8 board	144	7	5
over	Porter-Duff "over" operator	280	49	18
popcnt	Count the number of "1" bits in a 16-bit word.	96	5	6

at dynamically changing functional unit sizes based on dynamically maintained width information [2].

Static techniques for inferring minimum bit-widths using *don't care* detection are prevalent in the logic synthesis community, for example [6]. This approach computes *satisfiability don't care* sets on a network of Boolean operators. Such an analysis operates at the bit (and not at the word level) and is significantly slower but more precise than our approach. These algorithms are exponential in complexity, and even heuristic methods cannot address benchmarks of the size we are analyzing. Our algorithm has worst-case quadratic complexity. We compared our algorithm to the Synopsis Synplify compiler, a commercial CAD tool, using the DCT benchmark from Section 4. Our analysis runs two orders of magnitude faster and generates circuits within 30% of the size obtained by Synopsis.

Most similar to our work is Razdan [11]. His analysis uses a ternary logic of 0, 1 and *don't know* (denoted in this paper by `x`); he also operates on strings of bits, and uses forward and backward analyses. Although he handles loop induction variables for loops with a statically know trip-count, he does not offer a complete solution for handling loop-carried dependences, where a lot of savings can be gained.

Babb et al. [1] suggest that width analysis can be performed by determining the maximum values that can be carried on the wires, for example by examining loop bounds. This technique is further investigated by Stephenson et al. in [13]. These techniques are orthogonal to ours. Our analysis would very likely combine well with this technique, because the results of one could be used to seed the starting point of the other one, in the same way we handle induction variables.

6 Conclusions

We have presented BitValue, a compiler algorithm which infers statically the values of the bits computed by a program. Trimming constant bits or unused bits can reduce the width of the computed values, enabling the compiler to use

narrow width functional units, which have become available in new architectures (e.g., MMX, reconfigurable functional units, and Application-Specific Instruction Processors).

BitValue can be used to analyze both C and DIL programs to significantly reduce the number of bits used to perform computations. We show that BitValue inference can determine that on average 14% of the most significant bytes (and 20% of the bits) computed are unnecessary for programs from MediaBench and SpecINT95. BitValue analysis can reduce the size of the programs synthesized for a reconfigurable architecture between three- and twenty-fold. Finally, using our algorithm we were able to increase the simulated performance of several MediaBench programs by more than 20% when run on a CPU with a reconfigurable functional unit. The algorithm we present is an essential ingredient in developing a compiler which will target sub-word parallel media extensions, low power extensions, or reconfigurable devices.

References

1. J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *IEEE/FCCM Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1999. MIT.
2. K. Bondalapati and V.K. Prasanna. Dynamic precision management for loop computations on reconfigurable architectures. In *IEEE/FCCM Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1999. Organization: University of Southern California.
3. D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA-5*, January 1999. Princeton University.
4. M. Budiu and S.C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *ACM/FPGA Symposium on Field Programmable Gate Arrays*, Monterey, CA, 1999.
5. M. Budiu and S.C. Goldstein. BitValue — Detecting and Exploiting Narrow Bitwidth Computations. Technical Report CMU-CS-00-141, Carnegie Mellon University, June 2000.
6. M. Damiani and G. de Micheli. Don't care specifications in combinational and synchronous logic circuits. In *IEEE Transactions on CAD/ICAS*, pages 365–388, 1992.
7. S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.
8. C. Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.
9. P. Marwedel and G. Goossens, editors. *Code generation for embedded processors.* Kluwer Academic Press, 1995.
10. A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

11. Rahul Razdan. *PRISC: Programmable reduced instruction set computers*. PhD thesis, Harvard University, May 1994.
12. http://www.specbench.org/osg/cpu95/.
13. M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.
14. E. Stoltz, M. P. Gerlek, and M. Wolfe. Extended SSA with Factored Use-Def chains to support optimization and parallelism. In *Proceedings Hawaii International Conference on Systems Sciences*, Maui, Hawaii, Jan. 1994.
15. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

General Matrix-Matrix Multiplication Using SIMD Features of the PIII

Douglas Aberdeen and Jonathan Baxter

Research School of Information Sciences and Engineering
Australian National University
Douglas.Aberdeen@anu.edu.au, Jonathan.Baxter@anu.edu.au

Abstract. Generalised matrix-matrix multiplication forms the kernel of many mathematical algorithms. A faster matrix-matrix multiply immediately benefits these algorithms. In this paper we implement efficient matrix multiplication for large matrices using the floating point Intel SIMD (Single Instruction Multiple Data) architecture. A description of the issues and our solution is presented, paying attention to all levels of the memory hierarchy. Our results demonstrate an average performance of 2.09 times faster than the leading public domain matrix-matrix multiply routines.

1 Introduction

A range of applications such as artificial neural networks benefit from GEMM (generalised matrix-matrix) multiply routines that run as fast as possible. The challenge is to use the CPU's peak floating point performance when memory access is fundamentally slow. The SSE (SIMD Streaming Extensions) instructions of the Intel Pentium III chips allow four 32-bit (single precision) floating point operations to be performed simultaneously. Consequently, efficient use of the memory hierarchy is critical to being able to supply data fast enough to keep the CPU fully utilised. In this paper we focus on the implementation of an efficient algorithm for the Pentium SIMD architecture to achieve large, fast, matrix-matrix multiplies. Our code has been nicknamed Emmerald.

Without resorting to the complexities associated with implementing Strassen's algorithm on deep-memory hierarchy machines [5], dense matrix-matrix multiplication requires $2MNK$ floating point operations where $A : M \times K$ and $B : K \times N$ define the dimensions of the two matrices. Although this complexity is fixed, skillful use of the memory hierarchy can dramatically reduce overheads not directly associated with floating point operations. It is the optimization of memory hierarchy combined with the SSE that gives Emmerald its performance.

Emmerald implements the SGEMM interface of Level-3 BLAS, and so may be used immediately to improve the performance of single-precision libraries based on BLAS (such as LAPACK [4]). There have been several recent attempts at automatic optimization of GEMM for deep-memory hierarchy machines, most

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 980–983, 2000.
© Springer-Verlag Berlin Heidelberg 2000

notable are PHiPAC [3] and the more recent ATLAS [6]. ATLAS in particular achieves performance close to optimized commercial GEMMs. Neither ATLAS nor PhiPAC make use of the SSE instructions on the PIII for their implementation of SGEMM.

Our experiments showed that ATLAS achieves a peak of 375 MFlops/s for single-precision multiplies on a PIII @ 450 MHz, or 0.83× *clock rate*. Our matrix-matrix multiply using SIMD instructions achieves a peak of 890 MFlops/s, or 1.98× *clock rate*. We also report a price/performance ratio under USD $1/MFlop/s for training Neural Networks for Japanese OCR. The following section will describe our novel use of the SSE for Emmerald, followed by a description of optimizations designed to improve use of the memory hierarchy. The paper concludes with a comparison between ATLAS and Emmerald.

2 SIMD Parallelization

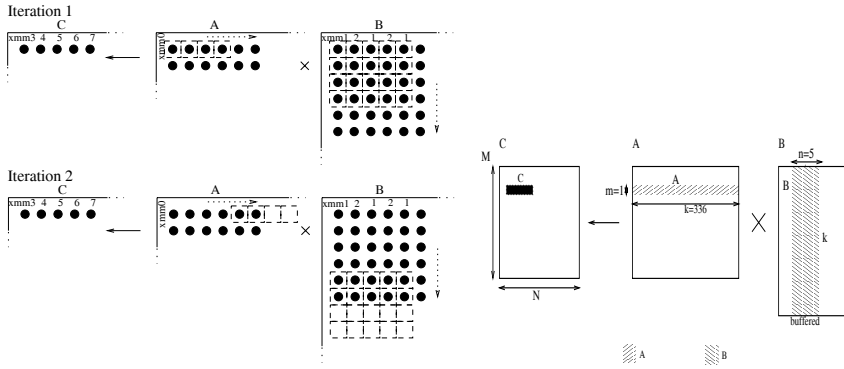
Two core strategies are employed to minimise the ratio of memory accesses to floating point operations: accumulate results in registers for as long as possible to reduce write backs and re-use values in registers as much as possible. In [2] several dot-products were performed in parallel as the innermost loop of the GEMM. We took the same approach and found experimentally that 5 dot-products in the inner loop gave the best performance. Figure 1(a) shows how these 5 dot products utilise SIMD parallelism.

Four values from a row of A are loaded into an SSE register. This is re-used five times by doing SIMD multiplication with four values from each of the five columns of B used in the inner loop. Two SSE registers are allocated to loading values from B. Results are accumulated into the remaining five SSE registers. When the dot-product ends each SSE result register contains four partial dot-product sums. These are summed with each other then written back to memory. For the best efficiency, the dot product length is maximised with the constraint that all data must fit into L1 cache.

3 Memory Hierarchy Optimizations

A number of standard techniques are used in Emmerald to improve performance. Briefly, they include:

- *L1 blocking*: Emmerald uses matrix blocking [2, 3, 6] to ensure the inner loop is operating on data in L1 cache. Figure 1(b) shows the L1 blocking scheme. The block dimensions m and n are determined by the configuration of dot-products in the inner loop (Section 2) and k was determined experimentally.
- *Unrolling*: The innermost loop is completely unrolled for all possible lengths of k in L1 cache blocks while avoiding overflowing the instruction cache.
- *Re-buffering*: Since B' is large (336×5) compared to A' (1×336), we deliberately buffer B' into L1 cache. By also re-ordering B to enforce optimal memory access patterns we minimise translation look-aside buffer misses [6].



(a) Five dot products using eight SSE registers (`xmm[0-7]`). Each circle is an element in the matrix. Each dashed square represents one floating point value in an SSE register.

(b) L1 blocking for Emmerald: $C' \leftarrow A'B'$ where A' and B' are in L1 and C' is accumulated in registers.

- *Pre-fetching*: Values from A' are not buffered in L1. We make use of SSE pre-fetch assembler instructions to bring A' values into L1 cache when needed.
- *L2 Blocking*: Efficient L2 cache blocking ensures that peak rates can be maintained as long as A, B and C fit into main memory.

4 Results

The performance of Emmerald was measured by timing matrix multiply calls with size $M, N, K = \{16, 17 \ldots 700\}$. To ensure a conservative performance estimate we use wall clock time on an unloaded machine rather than CPU time; the stride of the matrices (which determines the separation in memory between each row of matrix data) is fixed to the largest matrix size (700) and caches are flushed between calls to `sgemm()`. Timings were performed using a PIII 450 MHz with 128 MB RAM, 512 KB L2 cache and a 100 MHz bus.

Figure 2 shows Emmerald's performance compared to ATLAS and a naive three-loop matrix multiply. The average MFlop/s rate of Emmerald after size 100 is 1.69 times the clock rate of the processor and 2.09 times faster than ATLAS. A peak rate of 890 MFlops/s is achieved when $m = n = k = stride = 320$. This represents 1.97 times the clock rate. The largest tested size was $m = n = k = stride = 4032$ on a 550 MHz machine which ran at 1016 MFlops/s.

We have used Emmerald in distributed training of large Neural Networks with more than one million adjustable parameters and a similar number of training examples [1]. By distributing training over 196 Intel Pentium III 550 MHz processors, and using Emmerald as the kernel of the training procedure, we achieved a sustained performance of 152 GFlops/s for a price performance ratio of 98¢ USD/MFlop/s (single precision).

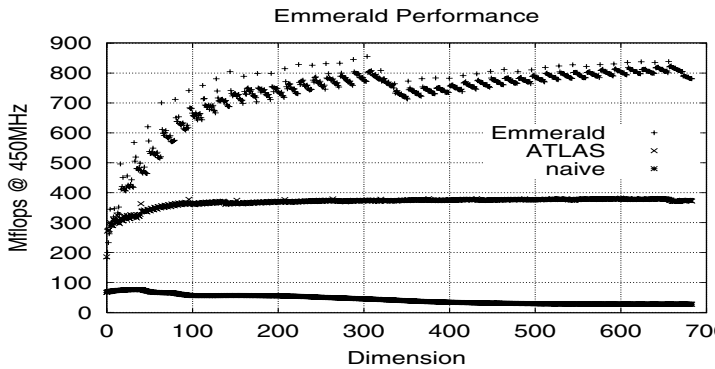


Fig. 2. Performance of Emmerald on a PIII running at 450 MHz compared to ATLAS SGEMM and a naive 3-loop matrix multiply. Note that ATLAS does not make use of the PIII SSE instructions.

5 Conclusion

This paper has presented Emmerald, a version of SGEMM that utilises the SIMD instructions and cache hierarchy of the Intel PIII architecture. An application demonstrating the cost-effectiveness of such work was also reported. This code and the full version of this paper is available from `http://beaker.anu.edu.au/research.html`.

References

[1] D. Aberdeen, J. Baxter, and R. Edwards. 98¢ /mflop, ultra-large-scale neural-network training on a PIII cluster. Sumbitted to SC2000, May 2000.

[2] B. Greer and G. Henry. High performance software on Intel Pentium Pro processors or Micro-Ops to TeraFLOPS. Technical report, Intel, August 1997. http://www.cs.utk.edu/ ~ghenry/sc97/paper.htm.

[3] J.Bilmes, K.Asanovic, J.Demmel, D.Lam, and C.W.Chin. PHiPAC: A portable, high-performace, ANSI C coding methodology and its application to matrix multiply. Technical report, University of Tennessee, August 1996. http://www.icsi.berkeley.edu/ ~bilmes/phipac.

[4] Netlib. *Basic Linear Algebra Subroutines*, November 1998. http://www.netlib.org/blas/index.html.

[5] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *Super Computing*, 1998.

[6] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. Technical report, Dept. of Computer Sciences, Univ. of TN, Knoxville, March 2000. http://www.cs.utk.edu/ ~rwhaley/ATLAS/atlas.html.

Redundant Arithmetic Optimizations

Thomas Y. Yeh and Hong Wang

[1]Intel Corporation, Microprocessor Research Lab,
2200 Mission College Blvd., Santa Clara, CA 95052
{Thomas.Y.Yeh, Hong.Wang}@intel.com

Abstract. Redundant arithmetic can be implemented on general-purpose processors to gain significant speedup. Redundant arithmetic speeds up the execution of certain dependent instructions by removing extra work and enabling higher frequencies. However, the issues of worst case delay, scheduling and power must be resolved. We will present the ECS representation format along with initial data and analysis on these critical issues.

1 Introduction

In general, redundant arithmetic speeds up dependent arithmetic computations by eliminating unnecessary conversions between intermediate and two's complement forms. Intermediate values during computations are expressed in redundant number representations that use multiple bits to indicate the value of a binary digit. By exploiting the dependency of certain operations, significant speedup can be obtained by removing sub-operations (conversions) and enabling a shorter cycle time. The carry-free characteristic of redundant arithmetic provides a scalable solution for addition of increased data path width. This optimization improves both latency and throughput of the processor [1].

1.1 Contributions

Several issues with implementation on microprocessors have been discussed in [2], [3], and [4]. In this paper, we present the novel concept of extended carry-save optimization to improve the performance of redundant arithmetic. Worst case delay, instruction scheduling and power issues are also discussed. Initial data on potential speedup with respect to in-order and out-of-order machines are presented. All operations for redundant arithmetic considered are integer operations.

2 Worst-Case Delay

Redundant arithmetic can be applied to add, subtract, multiply, divide, addressing, compares, and shift left operations while conversion back to two's complement form is

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 984-988, 2000.
© Springer-Verlag Berlin Heidelberg 2000

required for other instructions including logic, store, shift right, etc. However, the worst-case latency may be longer in a machine where redundant arithmetic is always performed before the conversion. With carry-save representation, this happens when adding two conventional operands when the only dependent instruction needs conventional operands.

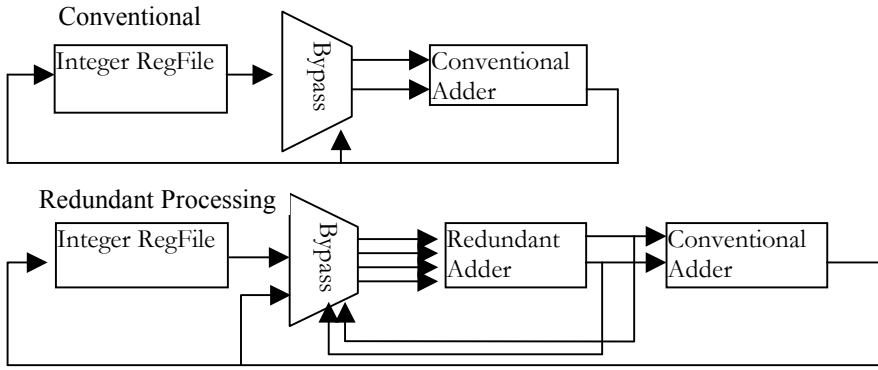


Fig. 1. Simple Redundant Arithmetic Implementation Block Diagram

Redundant addition is not necessary with two conventional operands, so the delay is increased. Overall speedup would require a longer chain of dependent instructions. One solution is to detect operand redundancy and optimize with the extended carry-save format (ECS). ECS allows add/subtract operations with 2 conventional operands to directly map the operands to the result bit-vectors while maintaining the same execution latency as redundant addition with 4-2 carry-save adders (CSA). This is due to the fact that a 4-2 CSA can compress 5 inputs of the same weight to a sum, a carry, and an intermediate carry. For the right-most bit position, ECS add/subtract operation with 4-2 CSA is able to compress the maximum of 7 input bits. The details of ECS operations are presented in [5].

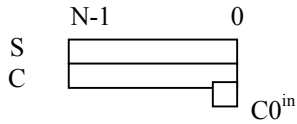


Fig. 2. Extended Carry-Save format (ECS)| N = operand width

To allow for fast addition, the approach is an extension of the radix-2 carry-save format. The direct mapping optimization can be used with the conventional carry-save format, but the $C0^{in}$ bit is needed for direct mapping of subtractions. As a result, the latency for add/subtract operations varies according to the operands' representation. Assuming the ratio of latencies between 64 bit conventional and redundant addition is two, ECS add/subtract operations result in 0 cycle for 2 conventional operands, 1 cycle for one or more redundant operands, and 2 cycles for conversion back to two's complement.

In ECS format, the S and C bit vectors are essentially 2 two's complement bit-vectors of precision N. Conversion requires conventional carry-propagate addition of the bit-vectors. To enable the ECS optimization, redundancy detection logic is needed to check the source of the operands.

3 Instruction Scheduling

The scheduling of instructions determines the distance in cycle time between dependent instructions. Redundant arithmetic optimizations can potentially change both the latency and the critical path for a given program. For long distances, redundant arithmetic is ineffective because the result of the producing operation could have been converted when the dependent operation is issued. The implications of this observation are different for in-order and out-of-order machines.

For out-of-order machines, the hardware dynamic scheduler can readily take advantage of latency changes for different instructions. The reorder buffer size limits the number of instructions that are examined when an instruction produces its result. Our simulation show that increasing the window size increases the percentage of windows with optimized critical paths. This shows that redundant arithmetic is effective in reducing the overall execution critical path.

With in-order machines, the compiler-scheduled code can also take advantage of redundant arithmetic. Dependency, operand redundancy, and conversion information would be extra parameters to be tracked by the compiler in generating code. As shown by our previous research using SPECint95, 22-29% of the instruction pairs on the critical path can be optimized by redundant arithmetic. This gives an upper limit on the percentage of critical instructions that can be enhanced with redundant arithmetic. However, existing binaries would require dynamic scheduling capability such as dynamic compilation to realize substantial benefit with redundant arithmetic.

4 Power

As discussed in [6], power consumption is a major performance-limiting factor in future processor designs. With redundant arithmetic, the number of signals increases to enable transfer of redundant data to and from the bypass, the register file, and the caches. In addition, the higher clock frequency enabled further increases the power.
However, redundant arithmetic can lead to a better performance/power ratio. Redundant adders are regular simple structures compared to complex conventional adders [7]. It increases the effective processing resource with minimal increase in logic. As shown by our simulation, less than 10% of all add/subtract operations on average needs conversion. This enables designs with fewer converters than redundant adders.

5 Simulation Data

To evaluate the effects of redundant arithmetic on the critical path, a simulator is developed to capture dependency information on dynamic instruction traces. The result representation of each operation is assigned based on instruction type and operand redundancy. Latency assignment and critical path capture utilizes a directed breadth-first search of the dependency graph.

The assumed machine model dictates the producers and consumers of redundant values. This model is based on past and current researches on redundant arithmetic including [8], [9], [10] and [11]. Producers include add/subtract and post-increment memory accesses while consumers include load/store, compare, and left shift operations. All memory accesses are assumed to hit the first level cache- completing in 1 original cycle. Logic operations complete in 1 redundant cycle.

Window sizes of 50, 100, and 300 were used along with sampling to gather data on the Perl, Go, and Compress benchmarks from the SPECint95 suite. Potential speedup is measured by comparing the longest redundant and. the longest original latencies. For optimized windows, the average speedup ranges from 30-50%.

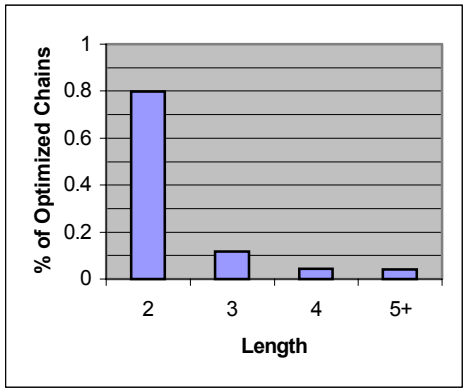


Fig. 3. Length of Optimized Chains

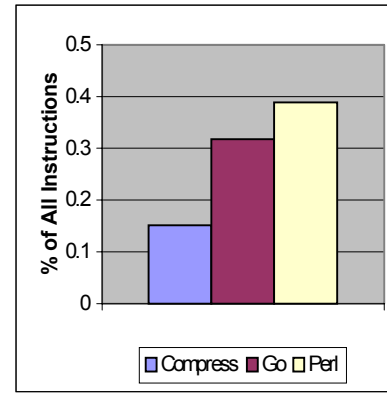


Fig. 4. % of Optimized Instructions

Compress shows potential for speedup with the high percentages of optimized compare and memory access instructions. The low percentage of optimized windows may be an effect of the 1-cycle memory access assumption.

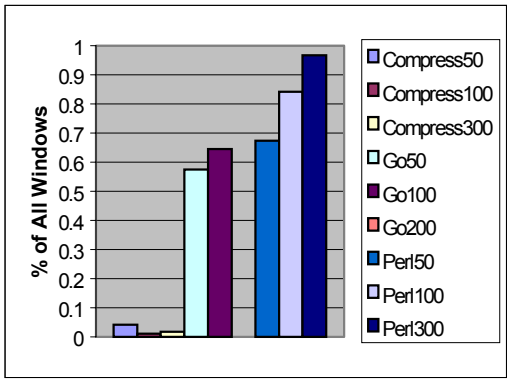


Fig. 5. Optimized Windows

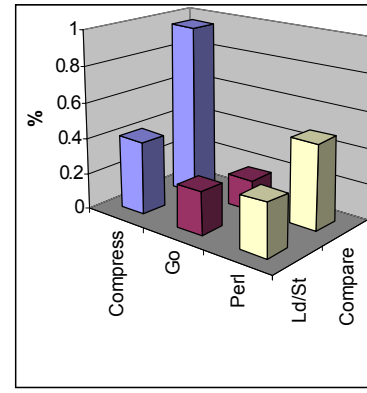


Fig. 6. % of Compare and MEM Optimized

References

1. Srinivas, H.R.; Parhi, E.K.: Computer Arithmetic Architectures with Redundant Number Systems. Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers, Vol. 1 (1994) 182-186
2. Glew, A.: Processor with Architecture for Improved Pipelining of Arithmetic Instructions by Forwarding Redundant Intermediate Data Forms. US Patent #5619664
3. Steiss, D.: Microprocessor Arithmetic Logic Unit using Multiple Number Representations. US Patent #5815420
4. Agarwal, R.; Fleischer, B.; Gustavson, F.: Recurrent Arithmetic Computation using Carry-Save Arithmetic. US Patent #5751619
5. Yeh, T.Y.: Integer Addition and Multiplication with Redundant Operands. UCLA 1999
6. Pollack, F.: New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. MICRO 1999 Keynote
7. Naffziger, S.: A Sub-Nanosecond 0.5 micron 64b Adder Design. Digest of IEEE International Solid-State Circuits Conference (1996) 362-363
8. Irita; Ogura; Fujishima; Hoh: Microprocessor Architecture Utilizing Redundant-Binary Operation. IEICE Transactions D-I, Vol. J81-D-I
9. Lynch, W.L.; Lauterbach, G.; Chamdani, J.I.: Low Latency through Sum-Addressed Memory. IEEE 1998
10. Lutz, D.R.; Jayasimha, D.N.: Early Zero Detection. 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '96 Proceedings 545-550
11. Cortadella, J.; Llaberia, J.M.: Evaluation of A+B=K Conditions Without Carry Propagation. IEEE Transactions on Computers, Vol. 41, No. 11 Nov 1992

The Decoupled-Style Prefetch Architecture

Kevin D. Rich and Matthew K. Farrens

University of California at Davis

Abstract. Decoupled processing seeks to dynamically schedule memory accesses in order to tolerate memory latency by prefetching operands. Since decoupled processors can not speculatively issue memory operations, control flow operations can significantly impact their ability to prefetch data. The prefetching architecture proposed here seeks to leverage the dynamic scheduling benefits of decoupled processing while allowing memory operations to be speculatively invoked. The prefetching mechanism is evaluated using the SPEC95 suite of benchmarks and significant reductions in cache miss rate are achieved, resulting in speed-ups of over 40% of peak for most of the inputs.

1 Introduction

Decoupled access/execute (DAE) architectures [6, 8, 7] rely on an intelligent compiler to separate the memory access and execution components of a program into independent instruction streams so they can be executed on separate processors. These processors are loosely coupled through architectural queues, which provide an asynchronous message passing channel. Using the queues to maintain the logical order of instructions, the two instruction streams are able to "slip" with respect to one another, allowing the access processor to lead the execute processor. This effectively migrates data fetching to a earlier time and produces what amounts to *dynamic* loop unrolling and pipelining.

Unfortunately, the strict FIFO nature of the queues limits the performance of decoupled architectures, since they cannot easily exploit branch prediction and speculative execution. In this paper an approach is proposed which makes use of the access instruction stream identification technology developed for the decoupled compiler to create an access stream which will prefetch data into the cache in order to reduce the occurrence of first-reference cache misses (a category of misses not generally addressed by current prefetching techniques.)

In the proposed architecture (D-SPA), the main processor (MP) is augmented with a *prefetch* processor (PFP), whose job is essentially that of decoupling's access processor - issue memory requests prior to the data being needed by the MP. However, the requirement that the data accesses be correct is removed since the prefetches only bring data into the data cache and do not effect MP state.

2 Background

Memory latency is not a new problem and many prefetching schemes have been explored to address it. Most of these proposals augment a uniprocessor with a

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 989–993, 2000.
© Springer-Verlag Berlin Heidelberg 2000

simple, parameterizable prefetch-engine. The compiler is responsible for analyzing the source code and identifying regular access patterns that can be described with parameters of the form `<start address>`, `<stride>`, and `<stop address>`. Examples of this include work by Chiueh [3], Chen [2], VanderWiel and Lilja [9].

The prefetch processor of D-SPA differs in that the hardware is considerably more powerful. Instead of having the prefetch engine execute a small set of parameterizable routines, the prefetcher is a fully functional integer-only processor — essentially the access processor from the DAE architecture. Like the access processor, it is loaded with its own executable. The prefetcher relies on the main processor only for flow control information and, when absolutely necessary, some data (function parameters and return values). Unlike the access processor, the prefetch processor only issues non-binding prefetches. The prefetched data is loaded into a cache or prefetch buffer, not into the processor's register file or a queue. Whether or not the data is ever used is solely a performance issue — it has no impact on program correctness.

3 The Decoupled-Style Prefetch Architecture

D-SPA was designed to take advantage of the compilation techniques developed for decoupled processing, while eliminating the restrictions on the execution model which prevent speculative execution. This paper presents only an overview of the architecture - for a more detailed description of the proposal and it's effectiveness please see [4].

Conceptually, D-SPA can be seen as a merging of a standard uniprocessor and the DAE architecture. It consists of a uniprocessor (MP) augmented with a prefetching processor (PFP). The PFP employs a subset of the same base RISC-ISA as the MP. The MP performs loads and stores in the usual manner, oblivious to the existence of the PFP. Branches are implemented using the <external branch>/<branch from queue> technique employed to co-ordinate branch decisions in decoupling, except that the MP hardware sends the results of every conditional branch to the branch queue, and all conditional branch operations on the PFP are `bfq` instructions. The PFP prefetch instructions are similar to typical load operations except that the data is brought into the cache or a prefetch buffer.

In addition to issuing memory requests purely for the purpose of prefetching, the PFP needs to access memory in order to reference variables that it requires to perform address calculations. In order to ensure that the PFP makes no references which could impact MP correctness, it must be restricted from writing to shared memory locations.

When considering how to enforce this restriction, it is important to recall that D-SPA is intended to be an augmentation of an existing RISC processor, not an entirely new processing model. As a result, modifying the memory system to provide exclusive access or other capabilities is not an option. Instead, the execution model must be modified to ensure that the PFP does not write shared/global variables. This can be accomplished by making the compiler responsible for ensuring that the target location of any PFP store is on the PFP's

stack; if this can not be guaranteed at compile time, the store is not included in the PFP executable. (To reduce the limitations on slip, each processor maintains a private run-time stack.) This may result in the PFP producing incorrect memory requests, but since all of the PFP's prefetches are non-binding these errors will not affect program correctness.

The ability to run uniprocessor binaries on D-SPA is achieved via a few minor modifications to the hardware necessary to address the external branch implementation and the addition of the copy queue. In order to run D-SPA binaries on an un-augmented MP the existence of copy operations in the D-SPA binary must be addressed. For a detailed description see [4].

By specifying the architectural and execution models to closely model those of decoupled processing, it is possible to leverage the compilation techniques developed for DAE processing. The most significant task is the identification of those instructions necessary to compute addresses — this must be done in order to generate the PFP instruction stream.

Function calls present perhaps the biggest challenge for the compiler. In order to accurately prefetch within a function which accepts parameters, it may be necessary for the PFP to have the parameter values which are calculated by the MP and must be communicated to the PFP via the copy queue. The PFP may also require a copy of the return value of a function in order to generate prefetch addresses (this is identified during def-use analysis). Therefore, the compiler must insert the copy operations necessary to communicate function parameters and return values from the MP to the PFP.

4 Results

The experiments performed use a combination of trace-driven and execution-driven simulation techniques, because it provides a reasonable first-order approximation without requiring extensive compiler modifications or the creation of a special multiprocessor simulator. The cycle-level *sim-outorder* SimpleScalar simulator was used to simulate the execution of the benchmarks [1].

Sixteen of the eighteen SPEC95 benchmarks were used in order perform the preliminary analyses of D-SPA (two of the eight integer benchmarks were omitted because of incompatibilities with the simulation environment.)

Due to disk space limitations, full runs of these benchmarks was not feasible; instead, a "window" of several million instructions was selected on which to perform the simulation, these windows were selected based on work performed by Skadron [5].

Two different cache configurations were employed. The first consisted of an 8K (256 blocks by 8 words/block) direct-mapped, single cycle, L1 cache backed by a 256K (1024 blocks by 16 words/blocks) 4-way set-associative, six cycle, L2 cache. The second consisted of a 16K (512 blocks by 8 words/block) direct-mapped, single cycle, L1 cache backed by a 256K (1024 blocks by 16 words/blocks) 4-way set-associative, six cycle, L2 cache.

The small cache sizes were used because the SPEC95 benchmarks generally exhibit very low data cache miss rates. Since the goal of this research is to

determine if number of data cache misses can be reduced, there needs to be some data cache misses. Future studies are planned with different applications and more representative cache configurations.

The memory latency used was 18 cycles for the first cache-to-main memory bus transfer unit and 2 cycles for each other unit in a single cache-block request. An additional consideration is the number of memory ports available. The default SimpleScalar configuration provides two memory ports; however, preliminary studies in which the two processors shared two ports indicated that performance suffered. The results presented in this chapter use a configuration with three memory ports, with two dedicated to the MP and one dedicated to the PFP.

Four different prefetching schemes were investigated. All four issue prefetches for loads of global data. The four possible combinations are derived by including or excluding prefetches for references to stack-based data, and including or excluding prefetches for store operations. The simulations were done using both the 8K and 16K cache configurations described previously. Full details of these simulations are available in [4] - in this paper there is only room for an overview of the results.

Ideally the PFP will be running (will "slip") far enough ahead of the MP to effectively prefetch, but not so far ahead that it adversely impacts cache performance. With unbounded slip, the PFP reduces cache misses (and as a result, achieves a large percentage of the peak speed-up) for many of the benchmarks. However, prefetching results in an *increase* in the cache miss rate for several others.

In order to determine if the behavior of D-SPA on the poor performing benchmarks is in fact attributable to excessive slip, a set of experiments with metered slip was performed. The slip metering approach employed here sets a threshold on the number of outstanding prefetches permitted. When this quantity is hit, the PFP stalls until it drops below the threshold. For these experiments, the slip was metered at 10, 25, 50, and 100 outstanding prefetches. With slip metered at 50, D-SPA achieves over 40% of peak speed-up on ten of the sixteen benchmarks, and between 25% and 40% of peak speed-up on three others.

The overall impact of metering slip was substantial. In the case of tomcatv for instance, the negative impact of prefetching can be directly attributed to excessive slip. With unmetered slip, prefetching results in increased execution time. When slip is metered at 50 outstanding prefetches, 40% of peak speed-up is achieved. For applu and hyrdro2d the impact of metering is more dramatic. Metering slip results in over 60% of peak speed-up being achieved for applu, while for hydro2d over 75% of peak speed-up is achieved. Metering has a lesser impact on apsi, fpppp, mgrid, and swim, but in all cases it significantly improves the performance over the unmetered case.

5 Conclusion

In order to exploit the capabilities of decoupled processing, while removing some of the performance constraining restrictions of its execution model, the

decoupled-style prefetch architecture (D-SPA) is proposed. D-SPA shares several characteristics with the decoupled processing model, including the use of branch and copy queues for communication between the processors. Unlike the decoupled model, it does not employ queues in its interface with memory. The elimination of the memory queues, and the use of non-binding prefetches, allows for the use of speculative execution in the PFP, reducing the slip limiting effect of conditional branch operations. By specifying D-SPA's architectural and execution model to closely resemble that of decoupled processing, the compilation techniques developed for decoupled processing can be employed in a D-SPA compiler. Despite the existence of the branch and copy queues, binary compatibility with an existing RISC instruction set architecture is accomplished with a modicum of effort.

D-SPA reduces the restrictions on slip by employing speculative execution and issuing non-binding prefetches. As a result, it may be possible for the PFP to slip too far ahead of the MP, adversely impacting performance. In order to counter this, techniques to constrain, or meter, slip were developed.

The experiments performed here have shown that D-SPA has the potential to significantly reduce the frequency of data cache misses. While these experiments are only a first order approximation of the performance of D-SPA, they indicate that further evaluation is warranted.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison, 1997.

[2] Tien-Fu Chen. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[3] T.-C. Chiueh. Sunder: A programmable hardware prefetch architecture for numerical loops. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 488–497, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[4] Kevin D. Rich. *Compiler Techniques for Evaluating and Extending Decoupled Architectures*. PhD thesis, University of California at Davis, 2000.

[5] Kevin Skadron. *Characterizing and Removing Branch Mispredictions*. PhD thesis, Princeton University, June 1999.

[6] James E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[7] N.P. Topham and K. McDougall. Performance of the decoupled ACRI-1 architecture: the Perfect Club. In *Proceedings of High Performance Computing - Europe*, 1995.

[8] Gary S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design*. PhD thesis, University of California at Davis, 1997.

[9] Steven P. VanderWiel and David J. Lilja. A compiler-assisted data prefetch controller. Technical Report ARCTiC 99-05, University of Minnesota, May 1999.

Exploiting Java Bytecode Parallelism by Enhanced POC Folding Model

Lee-Ren Ton[1], Lung-Chung Chang[2], and Chung-Ping Chung[1]

[1] Department of Computer Science and Information Engineering
National Chiao Tung University
No. 1001, Dashiue Rd., Hsinchu, Taiwan 30056, ROC
{lrton, cpchung}@csie.nctu.edu.tw
[2] Computer & Communications Research Laboratories
Industrial Technology Research Institute
Building 51, No. 195-11, Sec. 4, Jungshing Rd., Judung Jen, Hsinchu, Taiwan 31041, ROC
lcchang@ccl.itri.org.tw

Abstract. Instruction-level parallelism of stack codes like Java is severely limited by accessing the operand stack sequentially. To resolve this problem in Java processor design, our earlier works have presented stack operations folding to reduce the number of push/pop operations in between the operand stack and the local variable. In those studies, Java bytecodes are classified into three major POC types. Statistical data indicates that the 4-foldable strategy of the POC folding model can eliminate 86% of push/pop operations. In this research note, we propose an Enhanced POC (EPOC) folding model to eliminate more than 99% of push/pop operations with an instruction buffer size of 8 bytes and the same 4-foldable strategy. The average issued instructions per cycle for a single pipelined architecture is further enhanced from 1.70 to 1.87.

1 Introduction

Internet has become the most feasible means of accessing information and performing electronic transactions. Java [1] is the most popular language used over the Internet owing to its portability, compact code size, object-oriented, multi-threaded nature, and write-once-run-anywhere characteristics.

The performance of the stack-based Java Virtual Machine (JVM) [2, 3] is limited by true data dependency. A means of avoiding such a limitation, i.e. stack operations folding, was proposed by Sun Microelectronics [4, 5, 6, 7] with folding capabilities of up to 2 and 4 bytecode. While executing, pre-defined and pre-stored folding patterns are compared with bytecodes in instruction stream sequentially. Consequently, we call this kind of folding as fixed-folding-pattern matching. Other researchers [8, 9, 10] have also proposed the folding method of this fixed-matching type.

In our earlier study, we proposed a dynamic-folding-pattern matching named POC folding model [11]. All bytecode instructions are classified into three major POC (Producer, Operator, and Consumer) types. Table 1 lists the POC types and the 'O' type is further divided into four sub-types according to their execution behavior.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 994-997, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Table 1. POC Instruction Types

POC	Description	Occurrence
P	An operation that pushes constant or loads variable from LV to OS	47.14%
O_E	An operation that will be executed in execution units	10.87%
O_B	An operation that conditionally branches or jumps to target address	11.54%
O_C	An operation that will be executed in micro-ROM or trap	22.19%
O_T	An operation that will force the folding check to be terminated	3.96%
C	An operation that pops the value from OS and stores it into LV	4.29%

In the POC folding model, foldability check is performed by examining each pair of consecutive instructions. By applying the POC folding rules, the two consecutive bytecode instructions may be combined into a new POC type, which is used in further foldability check with the following bytecode instructions. Consequently, the POC folding model is quite different from the fixed-matching one because there is no fixed-instruction-pattern. The POC folding rules can be represented as a state diagram shown in Fig. 1. If the Ps are not consumed immediately by O or C type instructions, they will be issued sequentially.

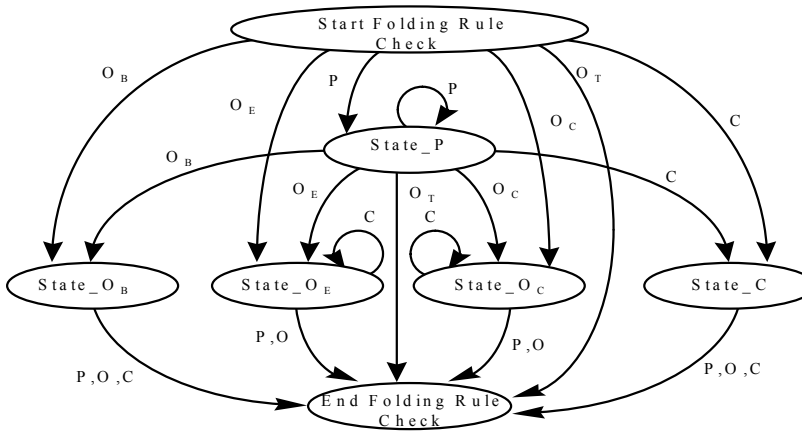


Fig. 1. Folding Rules for POC Model

2 Enhanced POC Folding Model

The main improvement of the EPOC model over the POC model is the capability of folding the discontinuous Ps. As shown in Fig. 2, the *P Counting* state will record how much Ps are there before the O or C type instructions. If there is no O or C type instruction in the instruction buffer, the Ps will be issued sequentially to the execution unit like the POC does. The *C Counting* state will check whether the preceding state is O_E state or *P Counting* state. If the preceding state is O_E, *C Counting* state will fold

the Cs into O_E. If it is *P Counting* state, then Ps are folded into Cs according to the number of Cs. Otherwise, if the C type instruction is the first instruction in instruction buffer, the EPOC issues the C sequentially.

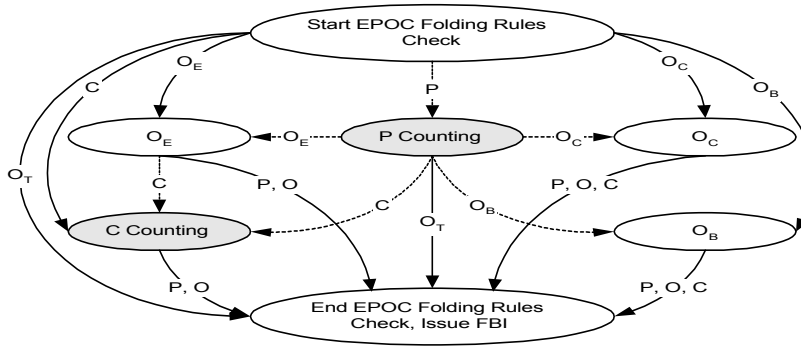


Fig. 2. Folding Rules for EPOC Folding Model

3 Performance Comparison of Various Folding Models

In Fig. 3, the average percentages of eliminated P + C type instructions for different foldability are shown. Note that the picoJava-II has the foldability of four according to the released specification. We duplicate the picoJava-II's performance results to each column for comparison only. The issued instructions per cycle (IIPC) for a single pipelined picoJava-II architecture for each model is shown in Fig. 4.

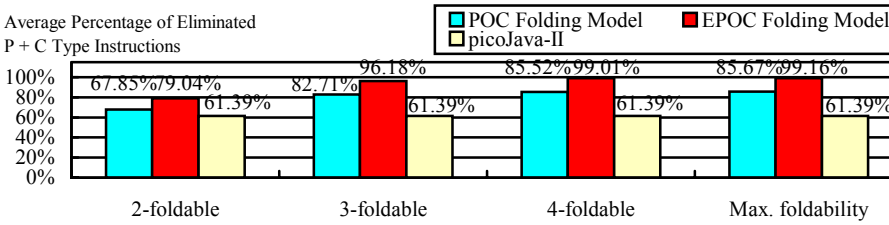


Fig. 3. Average Percentages of Eliminated P + C Type Instructions

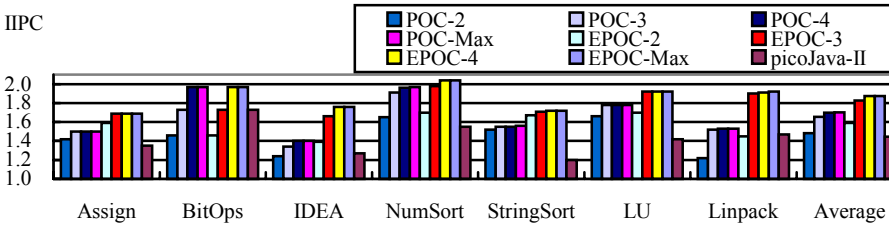


Fig. 4. Issued Instructions per Cycle for Each Folding Model

4 Conclusion

In this research note, we have proposed the EPOC folding model based on the previously proposed POC folding model. The dynamic-folding-pattern matching of POC and EPOC overrides the fixed-folding-pattern matching used in picoJava-II with the folding ratio of 39% and 61%, respectively.

The performance enhancement from POC to EPOC folding model benefits mainly from the foldability of discontinuous P type instructions, which results 85.5% and 99.0%.folding ratio using the 4-foldable strategy, respectively. That is, 44% and 50.9% program codes are folded, respectively.

The hardware implementation of the recursive EPOC folding model is integrated into decoding stage using parallel priority encoders to generate the source and destination fields of the FBI in constant delay time (non-recursive). Extra circuits must be added to maintain the instruction buffer after folding. For a superscalar Java processor of our current research, the EPOC folding model is integrated with the stack reorder buffer. Furthermore, the source-ready FBIs can be issued in parallel to exploit higher ILP.

References

1. James Gosling, Bill Joy and Guy Steele: The Java™ Language Specification, Addison-Wesley, Reading MA (1996)
2. Tim Lindholm and Frank Yellin: The Java™ Virtual Machine Specification, Addison-Wesley, Reading MA (1996)
3. Venners, B.: Inside the Java Virtual Machine,McGraw Hill, New York (1998)
4. M. O'Connor and M. Tremblay: picoJava-I: The Java Virtual Machine in Hardware, IEEE Micro, Vol. 17, No. 2, (1997) 45-53
5. H. McGhan and M. O'Connor,: picoJava: A Direct Execution Engine for Java Bytecode, IEEE Computer, (1998) 22-30
6. Sun Microsystems Inc.: picoJava-II™ Microarchitecture Guide, Sun Microsystems, CA USA (1999)
7. Sun Microelectronics: microJava™-701 Processor," http://www.sun.com/microelectronics/microJava-701/
8. Han-Min Tseng, et. al.: Performance Enhancement by Folding Strategies of a Java Processor, Proceedings of International Conference on Computer Systems Technology for Industrial Applications – Internet and Multimedia, (1997)
9. Lee-Ren Ton, et. al.: Instruction Folding in Java Processor, Proceedings of the International Conference on Parallel and Distributed Systems, (1997)
10. N. Vijaykrishnan, N. Ranganathan and R. Gadekarla: Object-Oriented Architectural Support for a Java Processor, Proceedings of the ECOOP'98, Lecture Notes in Computer Science, Springer Verlag, (1998)
11. L. C. Chang, L. R. Ton, M. F. Kao and C. P. Chung,: Stack Operations Folding in Java Processors, IEE Proceedings on Computer and Digital Techniques, Vol. 145, No. 5, (1998)

Cache Remapping to Improve the Performance of Tiled Algorithms

Kristof E. Beyls* and Erik H. D'Hollander

University of Ghent
Department of Electronics and Information Systems
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium

Abstract. With the increasing processing power, the latency of the memory hierarchy becomes the stumbling block of many modern computer architectures. In order to speed-up the calculations, different forms of tiling are used to keep data at the fastest cache level. However, conflict misses cannot easily be avoided using the current techniques. In this paper cache remapping is presented as a new way to eliminate conflict as well as capacity and cold misses in regular array computations. The method uses advanced cache hints which can be exploited at compile time. The results on a set of typical examples are very favorable and it is shown that cache remapping is amenable to an efficient compiler implementation.

1 Introduction

With Moore's Law still doubling the performance in 18 months, there is almost no limit to the processing power for the foreseeable future. Many performance programmers know that this is not the case, due to the speed gap between the processor and the memory. In fact, where the processor speed gains about 67% per year, the memory lags behind with only a gain of about 5-10%[9]. Using the same reasoning as Moore, one quickly finds out that a similar law says *the memory speed with respect to the processor halves each 22 months...* . From this observation, the growing importance of L1, L2 and L3 caches becomes evident and the objective is to keep the used data in the cache all the time.

Tiling[1, 15] is a well known method to improve the reuse of cached data in numerical applications by shortening the distance between the use and reuse of array elements. Tiling algorithms successfully eliminate capacity misses and therefore increase the cache hit ratio. However, the low associativity of caches may lead to a high number of conflict misses and slow down execution so that only a fraction of the attainable performance is obtained. Additional fine tuning of the tiling transformation is needed to reduce the conflict misses[2, 6, 8, 11, 13].

* Research financed by the Flemish government under contracts IWT-SB/991147 and GOA-12.0508.95)

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 998–1007, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In this paper, cache remapping is offered as a new technique to eliminate conflict misses in tiled algorithms. In addition, cache remapping produces no capacity misses and also cold misses are avoided for all but the first iteration.

Cache remapping is based on a dynamic rearrangement of the data at run time. During the execution of a loop, a parallel remap thread running concurrently with the original program thread relocates the tiled data needed by future iterations. The cache is split in two regions. One region contains all the data in the tile currently being processed, enabling the calculations to continue without memory stalls. At the same time the remap thread copies the data of the next tile into the other cache region, using proper address relocation and cache bypass. When the calculations have completed processing a tile, the original processing thread can immediately continue processing the next tile as it is already brought in the cache by the remap thread.

Section 2 explains cache remapping in detail. In section 3, experimental results are presented. Section 4 compares the techniques and the results with related work.

2 Cache Remapping Technique

2.1 Tiled Loop Nests

Fig. 1 shows a loop nest and the corresponding tiled loop.

Definition 1. *Tiling[15] transforms an n-deep loop nest into a $2n$-deep loop nest. The* tiled *loops in the resulting tiled loop nest are the n inner loops. The* tiling *loops are the n outermost loops. A loop nest will be notated as $\mathcal{L}$. The tiled and the tiling loops for $\mathcal{L}$ are $\mathcal{T}d(\mathcal{L})$ and $\mathcal{T}i(\mathcal{L})$ respectively. An* iteration tile *is the iteration space traversed by $\mathcal{T}d(\mathcal{L})$. The part of an array that is referenced during the execution of an iteration tile is a* data tile. *A* tile set *is the union of the data tiles of all the arrays accessed during an iteration tile execution.*

```
do i=1,N,1
  do j=1,N,1
    do k=1,N,1
      H(i,j,k)
```

(a) Original loop nest

```
L  Ti(L) do II = 1,N,B1
           do JJ = 1,N,B2
             do KK = 1,N,B3
               Td(L) do i = II,min(II+B1-1,N),1
                       do j = JJ,min(JJ+B2-1,N),1
                         do k = KK,min(KK+B3-1,N),1
                           H(i,j,k)
```

(b) Tiled loop nest

Fig. 1. A tiled loop nest

2.2 Cache Memory

For the development of the cache remapping technique, a cache is represented by a tuple (C_s, N_s, k, L_s)[3]

Definition 2. *The* cache size (C_s) *defines the total number of bytes in the cache. The* line size (L_s) *determines how many contiguous bytes are fetched from memory on a cache miss. A* memory line *refers to a cache-line-sized block in the memory which is aligned such that the data in it map into the same cache line. A* cache set *is the collection of cache lines in which a particular memory line can reside.* N_s *denotes the number of cache sets in a cache.* Associativity (k) *refers to the number of cache lines in a cache set. These parameters are related by the equation* $C_s = N_s \times k \times L_s$.

The start address A of a memory line determines the cache set N it maps to:

$$N = \left\lfloor \frac{A}{L_s} \right\rfloor \bmod N_s \tag{1}$$

A replacement algorithm decides the cache line in set N a memory line is copied to. In the rest of this paper the least recently used (LRU) replacement policy is assumed.

Definition 3. *Consider the memory lines accessed during the execution of* $\mathcal{L}$. *Then* $\mathrm{Ml}(\mathcal{L}, N)$ *represents the set of memory lines which map to cache set* N.

2.3 Conflict Misses in Tiled Algorithms

Consider a tiled loop $\mathrm{Td}(\mathcal{L})$ and a cache set N. When $\#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) > k$, more than k memory lines must be placed in the same cache set N. Only part of the memory lines can reside in the cache at the same time, and conflict misses arise.

Cache remapping copies tile sets into a contiguous C_s-sized buffer. Because of (1), k memory lines in the buffer map to each cache set. So, $\forall N \in N_s$: $\#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) = k$ and no conflict misses arise.

2.4 High-Level View of Cache Remapping

Cache remapping adds a remap thread to the program, which executes concurrently with the original processing thread executing the tiled loop nest $\mathcal{L}$ (see fig. 2). These two threads can execute in parallel on processors with multiple functional units. (for further detail, see sect. 2.5).

Consider an iteration point i of $\mathrm{Ti}(\mathcal{L})$. The two threads work in a pipelined way(see fig. 3):

- The *processing thread* executes tile i.
- The *remap thread* copies data tile set $i+1$ into the cache. If there is written data of tile set $i-1$ in the cache, it is first copied back to main memory to make place for tile set $i+1$.

At most two consecutive tile sets are in the cache at the same time. Between iterations of $\mathrm{Ti}(\mathcal{L})$, the two threads synchronize.

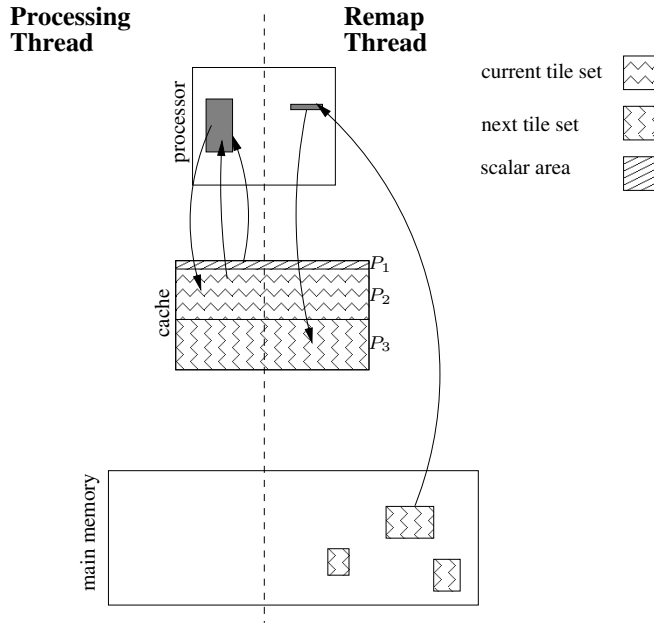


Fig. 2. The remap thread puts the next tile set in the cache while the original thread processes the current tile set. In the next phase, the processing and the remap thread will access P_3 and P_2 respectively.

Fitting the Current and Next Tile Set in the Cache The process thread accesses two kinds of variables in the memory: scalar variables which do not fit into the registers and arrays. To ensure that all data referenced by the process thread is in the cache, it is logically divided in three partitions: P_1, P_2 and P_3. P_1 is used to cache the scalars. P_2 and P_3 will each contain one tile set.

During the odd iterations of $\mathrm{Ti}(\mathcal{L})$, the process thread uses P_2, during the even iterations, it accesses P_3. The remap thread uses P_3 during the odd iterations and P_2 during the even iterations. It is clear that P_2 and P_3 must have the same size as they are used symmetrically.

Respecting Data Dependencies Problems arise when there are data dependencies between the tile sets of two consecutive iteration tiles.

If the process thread currently processes tile set i and writes into elements of tile set $i+1$, the remap thread prefetches these elements into the cache with the old values. When the process thread executes tile $i+1$, it will use the old values instead of the new.

A solution is to copy the new value of the shared elements to the cache partition the process thread will use. This must be done during the thread synchronization, which occurs between iterations of $\mathrm{Ti}(\mathcal{L})$.

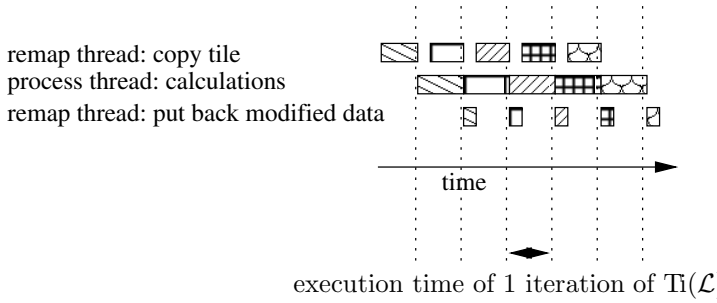


Fig. 3. The pipelined nature of cache remapping

2.5 Low-Level Details

Controlling Cache Behavior

Cache Shadow At the start of the program, a consecutive block of memory with size C_s is allocated and aligned on a memory line. We call this memory block the cache shadow. There's a one-to-one relation between the addresses in the cache shadow and the storage area in the cache. The area's P_1, P_2 and P_3 are allocated in this cache shadow.

To assure that the contents of the cache shadow always resides in the cache completely, cache hints are used. They make it possible to only cache the addresses in the cache shadow by bypassing the cache on memory references outside the cache shadow.

Cache Hints In modern instruction sets, cache hints[4, 5] are attached to load and store instructions. They specify if the referred data should be cached or not. When data is loaded/stored from/to P_1, P_2 or P_3, a cache hint tells the processor to cache the data. If an address outside the cache shadow is referenced, the cache hint tells the processor not to cache the data.

Thread Scheduling on a Single Threaded Superscalar Processor The process and remap threads need to run concurrently. Current microprocessors offer parallelism at the instruction level (ILP). This means that only nearby instructions in one thread can be executed simultaneously. To execute the remap thread and the process thread concurrently, these two threads need to be interwoven into a single thread at compile time. The instructions of the two threads must be interleaved so that the processor can execute instructions of the two threads during the same cycle.

On current processors, about a dozen functional units are present. Typically, data dependencies cause an average IPC (instructions per clock-cycle) no more than 2 to 3, so about ten functional units are left unused every clock-cycle. There are no dependencies between the remap and the process thread during the execution of Td($\mathcal{L}$). As a result, the remap thread can use the functional

units that are not used by the process thread. A good optimizing compiler can schedule the instructions of both threads so that they execute simultaneously.

Non-Stalling Memory Access The remap thread accesses main memory. Because the two threads are interwoven into one, it is important that the memory access doesn't stall the processor. When an instruction from the remap thread accesses main memory, there are enough independent instructions from the process thread ahead in the instruction stream to perform useful in-cache computations to overlap the latency.

Selection of Tile Size The tile size (B_1,...,B_n; $n = 3$ in the example) is chosen so that every tile set fits in P_2 and P_3. A large number of tile sizes satisfy this constraint. Let $iter_p = B_1 * \cdots * B_n$, the number of iterations executed by the process thread during a tile execution. Let $iter_r$ be the number of array elements that need to be remapped or put back during a tile execution. We choose to optimize the tile sizes so that the ratio $r = \frac{iter_p}{iter_r}$ is maximal. This choice assures that the processing power needed by the remap thread is as small as possible relative to the processing power needed by the process thread.

Loop Transformations and Thread Scheduling To lower the scheduling overhead, a number of loop transformations are performed to the loop nests in both threads. The remap thread originally consists of Q loop nests. Every loop nest remaps or puts back a data tile. Q_i^r is the number of elements that are remapped by loop nest i. Each of these loop nests are coalesced[10], and the body of the remaining loop is placed in an inlined remapping function (e.g. `remapA` in fig. 4(a)).

The innermost loop in the tiled loop nest $\mathrm{Td}(\mathcal{L})$ is unrolled $\lfloor r \rfloor$ times, then a remap call is inserted (see fig. 4(c)).

It is known at compile time how many times each remap function must be executed per iteration of $\mathrm{Td}(\mathcal{L})$. The outermost loop of $\mathrm{Td}(\mathcal{L})$ is split into Q parts. In each part, another remap function is called. The number of iterations in each part is chosen so that every remapping function is called at least Q_i^r times. So $Q_i^{B_1} * B2 * \left\lfloor \frac{B3}{r} \right\rfloor \geq Q_i^r$.

3 Implementation and Results

3.1 Processor Requirements

Three conditions must be met to enable cache remapping:

1. the processor provides the possibility to load data from main memory without bringing it into the cache, e.g. using cache hints,
2. multiple instructions execute concurrently, e.g. a superscalar processor,
3. the processor does not stall on a cache miss, as long as independent instructions are available in the instruction stream. This can be achieved using speculative loads[4].

The IA-64 architecture satisfies these requirements as well as all Explicitly Parallel Instruction Computing (EPIC)-style architectures.

```
remapA(int iter,A,p) {
  i1 = de_coalesce(iter);
  i2 = de_coalesce(iter);
  remap(p+i1*B2+i2,
        A[i1+II,i2+JJ]);
}
```

(a) One of the Q functions that remap one element

```
remap(double* x, double* y)
{
   fld.nta  r1,y
   fst.t1  x,r1
}
```

(b) The remap function. The `nta` cache hint means "don't cache", the `t1` cache hint means "cache into L1".

```
swap(p2,p3)
iter=0
do i = II,II+Q_1^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      H(i,j,k,p2) /* body r */
      ... /* times unrolled */
      H(i,j,k+r-1,p2)
          /* remap code */
      remapA(iter++,A,p3)
iter=0
do i = II+Q_1^{B_1},II+Q_1^{B_1} + Q_2^{B_1}-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      ...
      remapB(iter++,B,p3)
...
```

(c) The transformed tiled loop nest.

Fig. 4. The program transformations to efficiently interweave and schedule both threads into one. `p2` and `p3` are the start addresses of P_2 and P_3 respectively. It is assumed that — after inlining — the instruction scheduler will move enough independent instructions between both instructions in `remap` to allow useful computations during the main memory access.

3.2 Simulation

Since EPIC processors are not yet available, the Trimaran simulator[14] was used to simulate the behavior of the processor. The cache behavior was modeled by the well known Dinero cache simulator.

The experiment is a tiled matrix multiply executed on a processor with a 2-level cache. The L1 cache is 16Kb direct mapped with 32 byte lines. The L2 cache is a 256Kb 4-way set associative with 64 byte lines. We assume that the access latency of the L2 cache is 20 clock cycles and the access latency of the main memory is 65 clock cycles.

The cache remapping technique was compared with the original algorithm, a naively tiled algorithm not considering limited cache associativity and three optimized tiling algorithms, namely padding[8], copying[13] and LRW[6]. Each algorithm was coded, compiled and simulated for matrix dimensions between 20 and 400. For the cache remapping algorithm, the tiles on the border of the

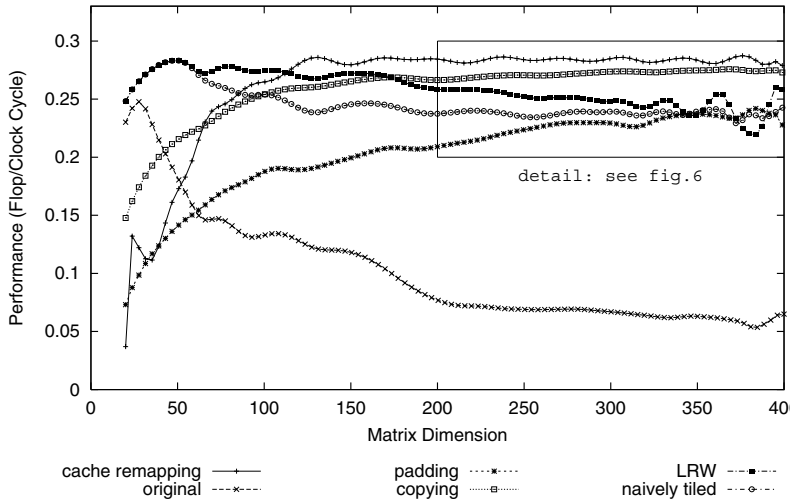


Fig. 5. Smoothed plot of the performance of several tiled matrix multiplications for dimensions 20 to 400. In this smoothed plot it is clear that the cache remapped algorithm outperforms the others for matrix sizes bigger than 150. A zoom of the actual performance plot can be found in figure 6.

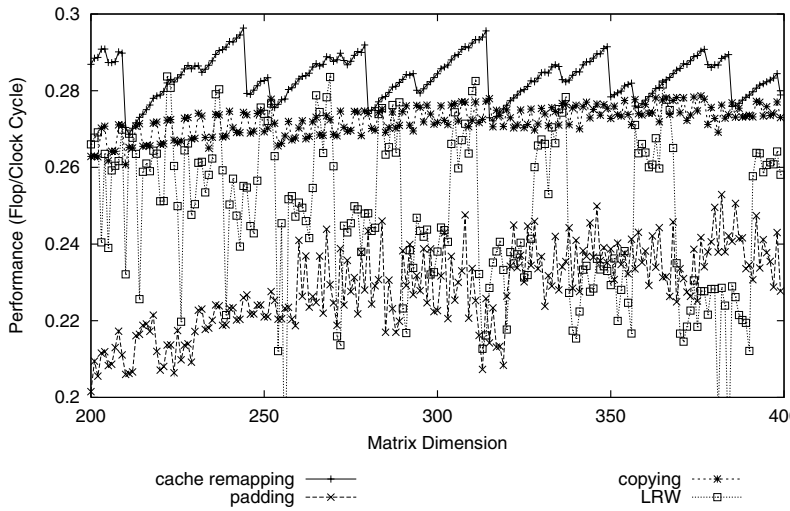


Fig. 6. The performance of cache remapping, padding[8], copying[13] and LRW[6] on matrix dimensions 200 to 400. The cache remapped algorithm has the same performance as the next best algorithm at worst. At best, a speedup of 10% over the next best algorithm is obtained.

iteration space were processed using the copying technique, because the pipelined nature of cache remapping suffers from processing tiles not completely filled with data.

The performance of the algorithms, expressed in number of floating point operations per clock cycle, is plotted in figure 5. Because the performance of some algorithms fluctuate, the data was smoothed using bezier curves to clearly visualize the trends. In figure 6 an exact plot is given for the four best algorithms for matrix dimensions 200 to 400. This plot shows that cache remapping always performs at least as good as the next best algorithm. At best, it yields a 10% speedup over the next best algorithm.

For matrix dimensions bigger than 150, cache remapping outperforms the alternative tiled algorithms. For matrix dimensions between 200 and 400, the average speedup compared to the second best algorithm (copying) is 5%. Compared with the original non-tiled algorithm, an average speedup of 4.5 is obtained.

4 Comparison with Related Work

Methods that select tile sizes to eliminate conflict misses[2, 6] sometimes result in small tiles, which reduce the performance. Padding[8] on the other hand uses large tile sizes and changes the data layout of the arrays by enlarging the dimensions with unused elements, in order to avoid cache conflicts. Unfortunately, this static adjustment cannot be optimized for every loop in a program simultaneously. Copying[13, 6, 11] eliminates conflict misses by copying the array tiles with the worst self interference to a contiguous buffer. Copying naturally involves overhead and the tradeoffs between copying and cache conflicts are discussed in [13].

In contrast to Padding and Tile Size Selection, cache remapping is independent of the array dimensions and doesn't require a change of the data layout. With respect to copying, cache remapping is able to cache tiles in a parallel thread, which runs concurrently with the processing thread. As a consequence, cache remapping has no conflict misses and incurs a minimal overhead.

The cache bypass and relocation technique was exploited by Lee[7] to use the cache as a set of vector register on i860 processors mimicking Cray's strided get/put[12]. Yamada[16] proposed prefetching and relocation by extending the hardware with a special data fetch unit which enables prefetching strided data without cache pollution. Our technique also combines cache bypass and relocation, but isn't limited to strided data patterns which allows it to prefetch and relocate data structures with non-constant strides such as data tiles.

5 Conclusion

The Von Neumann bottleneck nowadays hinders even a single processor. Cache remapping represents a promising technique to bridge the steadily growing gap between processor and memory speeds. It favorably compares with existing tiling

techniques and it uses the concepts of a new generation of processors. In future work the presented technique will be embedded in a EPIC compiler.

References

[1] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings, Supercomputing '92*, pages 114–124. IEEE Computer Society Press, November 1992.

[2] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN'95: conference on programming language design and implementation*, pages 279–290, June 1995.

[3] S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999.

[4] Intel. *IA-64 Application Developer's Architecture Guide*, May 1999.

[5] G. Kane. *PA-RISC 2.0 architecture*. Prentice Hall, 1996.

[6] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California*, pages 63–74, April 1991.

[7] K. Lee. The NAS860 library user's manual. Technical report, NASA Ames Research Center, Moffett Field, CA, March 1993., 1993.

[8] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers*, 48(2):142–149, Feb 1999.

[9] D. Patterson. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March-April 1997.

[10] C. D. Polychronopoulos. Loop coalesing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242, Pennsylvania, Pa, USA, Aug. 1987. Pennsylvania State Univ. Press.

[11] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction (CC'99)*, March 1999.

[12] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. ASPLOS VII*, Cambridge, MA, Octobe 1996.

[13] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In IEEE, editor, *Proceedings, Supercomputing '93*, pages 410–419, March 1993.

[14] Trimaran. *The Trimaran Compiler Research Infrastructure for Instruction Level Parallelism*. The Trimaran Consortium, 1998. http://www.trimaran.org.

[15] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[16] Y. Yamada, J. Gyllenhaal, G. Haab, and W. mei Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 118–127, San Jose, California, Nov. 30–Dec. 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

Code Partitioning in Decoupled Compilers

Kevin D. Rich and Matthew K. Farrens

University of California at Davis

Abstract. Decoupled access/execute architectures seek to maximize performance by dividing a given program into two separate instruction streams and executing the streams on independent cooperating processors. The instruction streams consist of those instructions involved in generating memory accesses (the *Access* stream) and those that consume the data (the *Execute* stream). If the processor running the access stream is able to get ahead of the execute stream, then dynamic pre-loading of operands will occur and the penalty due to long latency operations (such as memory accesses) will be reduced or eliminated.
Although these architectures have been around for many years, the performance analyses performed have been incomplete for want of a compiler. Very little has been published on how to construct a compiler for such an architecture. In this paper we describe the partitioning method employed in *Daecomp*, a compiler for decoupled access/execute processors.

1 Introduction

Program execution can be viewed as a two-part process — the moving of data to and from memory and the performing of some operation on that data. Conceptually, these two steps can be represented by two cooperating processes, the *memory access* process and the *computation* (or *execute*) process.

A decoupled architecture seeks to achieve high performance by running these two processes on separate cooperating processors, allowing out-of-order execution between the two instruction streams. The processors both traverse the same dynamic flow graph, though not necessarily at the same pace. To allow the processors to execute different portions of the graph, architecturally visible queues are employed to buffer information produced by one process for consumption by the other. If the access process can run sufficiently ahead of the execute process on this flow graph then it will dynamically preload the operands consumed by the execute process and hide the latency of memory access operations. If this occurs it is said that access process has *slipped* with respect to the execute process, or that *slip* has been achieved.

The execution of the two processes on separate processors provides a simple mechanism for supporting limited out-of-order execution, and the ability to issue more than one instruction per cycle. Because of its simplicity and potential ability to tolerate long memory latencies, there is a continuing interest in decoupled architectures [8, 9, 10, 3, 12]. In particular, decoupled architectures

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1008–1017, 2000.
© Springer-Verlag Berlin Heidelberg 2000

may be of great interest in the growing field of power-conscious processor design because the simplified method of exploiting ILP does not require many of the large, power-hungry circuits needed by superscalar designs.

While decoupled processing has existed in various incarnations for years [7, 10, 1, 9] there has been very little published on the compilation techniques necessary. The most fundamental compilation issue is how the instructions/operations will be allocated or assigned to the access and execute processors. This process is referred to as the *partitioning* of the code. In the rest of this paper we describe the partitioning scheme employed by Daecomp, an ANSI-C compiler we developed to allow for a more complete analysis of decoupled processing. An example of actual compiler output will be shown, as will some simulation results obtained using Daecomp-produced code.

2 Background

Despite the appearance of decoupled architectures in the literature for many years, very little information is available regarding the techniques necessary for a decoupled compiler. At least two functioning decoupled compilers exist that we are aware of, but both were for commercial products (the ZS-1 [7] and the ACRI-1 [9]) and therefore details of the compiler construction have not been published.

The research most closely related to the work we are presenting here was performed by Topham et al. [8] in which they investigate source-code level transformations that can be made for the ACRI-1 with the goal of reducing the frequency of AP-EP synchronization. Our work focuses on the lower-level partitioning task performed by the compiler.

3 Processor Model

The two cooperating instruction streams produced by a decoupled compiler will execute on two separate processing elements, which communicate in a message passing manner via architecturally visible queues. A block diagram of the target architecture is shown in Figure 1. Memory addresses are sent to memory via the Load and Store Address Queues (LAQ,SAQ), and memory operands are received or sent via the Load and Store Data Queues (LDQ,SDQ). In addition to the LAQ-LDQ and SAQ-SDQ queue pairs, each processor in this model has a complete set of *alternate* memory queues. These queues allow the processor to perform loads and stores on its own behalf (*self-loads* and *self-stores*). There are also Copy Queues (to allow data transfer between the processors) and Branch Queues (to provide control flow synchronization).

Use of these queues allows instructions from the two instruction streams to slip with respect to one other, providing dynamic scheduling and out-of-order issue capabilities without requiring the architectural complexity of a superscalar processors instruction window. Decoupling seeks to tolerate memory latency via this dynamic scheduling, in particular via the early issue of memory operations.

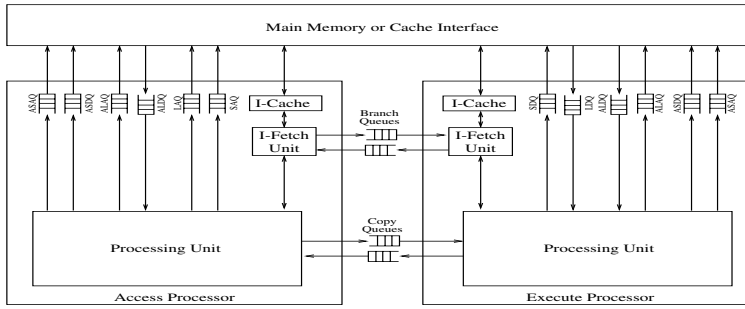


Fig. 1. Decoupled Access/Execute Processor - Conceptual Diagram

4 The Compiler

Daecomp is based on *cbend*, a multi-threaded compiler for the Concurro architecture produced by Bernard K. Gunther. The multi-threaded aspects of cbend were stripped out and the code specific to decoupled compiling was added. In addition, cbend's register allocator and code emission routines underwent modifications, primarily to deal with the use of queues.

Aside from partitioning the code, perhaps the biggest challenge encountered during compiler construction was handling function calls efficiently and correctly. Since the compiler is responsible for parameter set-up on the stack and/or in the parameter registers (each processor has its own register file), one major design decision is whether each processor will have a private copy of the run-time stack or if a single stack will be shared.

The decision impacts how function calls are managed, requiring different protocols for parameter passing and different considerations for local (i.e., stack-based) variable management. Passing parameters to a function under a single shared stack model is simpler than under a private stack model, but since the stack based variables are a shared resource there are a variety of potential performance ramifications. For performance reasons a dual-stack model was initially explored — unfortunately it proved to have a fatal flaw related to passing pointers to pointers so the partitioning technique detailed here assumes a single, shared run-time stack.

5 Code Partitioning

The task of the compiler is to partition a directed acyclic graph into two separate, cooperating, decoupled instruction streams based on def-use information. The graph consists of *nodes*, which represent the instructions/operations to be performed, and *edges*, which convey the dependencies between nodes. The goal is to produce a partitioning that reasonably balances the processor workload and

makes it possible for the two instruction streams to slip with respect to each other.

In the partitioning technique employed by Daecomp, each operation is assigned to a processor based on the class of operation (e.g., address generation, function parameter set-up, etc.) to which it belongs.

To start the marking process operations like loads and stores are designated as *anchors*. Anchor nodes are either *sinks* or *sources* of expressions. Sink nodes represent operations (e.g., stores) upon which no other operation directly depends, while source nodes represent operations which do not depend directly on any other operation (e.g., dequeuing a data value). Once the anchors are identified, the graph is again traversed, this time assigning (or *marking*[1]) instructions which depend on an anchor (or on which an anchor depends) to the same processor as the anchor. Interprocessor data dependencies are handled by inserting copy instructions from one processor to the other where necessary.

Once the partitioning (detailed below) is completed the compiler has created two graphs, one for each processor. Register allocation and code emission are performed on each graph resulting in the decoupled object code file. The five-step partitioning process will now be detailed.

Step 1: Anchor Return Value Usage

To minimize inter-processor copies it is desirable to anchor a function's return value calculation on the processor which is going to use the value. Unfortunately, a function may be called from several different locations within the program (or programs, in the case of a library routine) and the return value may be needed on different processors depending upon the calling location. Additionally, it is unlikely the compiler will have any information regarding the caller(s) of the function that it is compiling.

Therefore, it must be determined a priori which is the *return value processor.* The node which puts the return value into the return value register is marked for the return value processor and serves as an anchor. The compiler currently requires all return values to reside on the EP[2].

Step 2: Split Loads and Stores

The splitting of memory accesses into address and data portions lies at the heart of decoupled processing. Load operations are split into two parts: the operation that enqueues the load address onto the load queue, and the operation that dequeues the loaded data from the load data queue. Stores are similarly split into the operation which enqueues the store address and the operation which enqueues the store data.

[1] A node which is *marked* has been assigned to a processor. A node which has not yet been assigned to a processor is *unmarked.*

[2] In the future, the compiler will be modified to handle return values more intelligently. For a function which does not return a pointer it will assume that the value computed is needed by the EP and thus the EP will be the return value processor. Pointer valued functions are most likely computing a value related to an address calculation, so the AP will be the return value processor if the return value is a pointer.

```
UNIPROCESSOR          ⇒   ACCESS PROCESSOR      EXECUTE PROCESSOR
load  rA, rB, 8           add   LAQ, rB, 8      move  rA, LDQ
store rX, rY, 8           add   SAQ, rX, 8      move  SDQ, rY
```

As indicated in the above example, the node which enqueues the load or store address is assigned to the AP and the node which enqueues or dequeues the data is marked for the EP; these nodes serve as anchors. For the load operation, the node for the instruction which dequeues from the LDQ is a source node and the node which enqueues the load address on the LAQ is a sink. For the store operation, both the node for the instruction which enqueues the address on the SDQ and the node which enqueues the data on the SDQ are sinks.

It is important to note that while the processors may be identical, the EP cannot be allowed to enqueue addresses for data which the AP can access (e.g., global data or data on the AP's stack), because there is the potential for the violation of RAW, WAR, or WAW dependencies. So *functionally* speaking such *reverse decoupled* loads and stores are not permitted. In the shared stack model these restrictions result in the EP not be being permitted to use its alternate memory queues.

Step 3: Propagate Load/Store Processor Markings

Propagation of markings entails starting at each of the anchors operations and traversing the (sub)graph rooted at that anchor, assigning all unmarked data-dependent nodes in the graph to the same processor as the anchor. The propagation occurs in two directions, down from a sink and up from a source. When propagating the markings down from a sink, the marking propagates down through its children to all of its descendants. When propagating the marking up from a source, the marking propagates up through its parents to all of its ancestors.

If during the traversal a node is encountered that is marked for the other processor, a copy is necessary in order to communicate the value from one processor to the other. In the below example of an AP-to-EP copy, AP line 2 enqueues the value on the copy queue, and EP line 1 dequeues the value from the copy queue.

```
        ACCESS PROCESSOR              EXECUTE PROCESSOR
1:      add   rA, rB, 8               sub  rY, CPQ, rX
2:      move  CPQ, rA
```

The compiler attempts to avoid EP-to-AP copy operations as they introduce slip-reducing dependencies. The fact that a copy is required means that the value is needed by both processors and thus the computing expression is a common sub-expression (CSE). If the CSE is inexpensive to compute, *common sub-expression replication* may be employed (in which both processors perform the calculation of the sub-expression). If the CSE is too expensive to replicate, but cheap enough that moving it to the AP would not lead to grossly unbalanced code, it is moved to the AP (it is *stolen*) and an AP-to-EP copy is inserted. Allowing the AP to steal the sub-expression eliminates the need for the slip-reducing

EP-to-AP copy. If this *common sub-expression stealing* would result in code balancing problems, or is otherwise infeasible, an EP-to-AP copy is inserted. The compiler makes these decisions by estimating the computation cost of the sub-expression and comparing it to threshold values[3]. Copies related to load and store operations may also be eliminated by converting standard decoupled loads or stores into self-loads or self-stores on the access processor.

If an EP-to-AP copy is unavoidable, the compiler performs code scheduling in order to minimize the impact of the copy. The EP's enqueuing operation is placed as early as possible in its instruction stream, and the AP's dequeuing operation is placed as late as possible in its instruction stream. This technique is used extensively with function parameters.

Step 4: Branch Splitting

After marking propagation, conditional branch operations are split into their two cooperating counterparts, the *external branch* operation and the *branch from queue* operation. An external branch is a conventional branch which also writes the branch decision (taken/not taken) to the processor's outgoing branch queue, while a branch from queue (`bfq`) is a conditional branch that is taken or not based on the value at the head of the processor's incoming branch queue. This mechanism allows the processors to traverse the flow graph in identical fashion by easily synchronizing control flow decisions. In the below example the `x` in the `brxnz` indicates an external branch.

```
UNIPROCESSOR          ⇒   ACCESS PROCESSOR       EXECUTE PROCESSOR
cmpne rC, rA, rB          cmpne rC, rA, rB
brnz  rC, @L1             brxnz rC, @L1          bfq   @L1
```

This splitting is only performed on conditional branches. Unconditional flow control operations (e.g., jumps, function calls) are simply duplicated on each of the processors and do not need to be split.

The compiler must determine which processor is to receive each of the two branch operation components. There are several issues to be considered when making this decision — for example, if the `bfq` is put on the AP then a slip reducing AP-on-EP dependency is introduced. In order to avoid such dependencies, the compiler makes every attempt to put the comparison and the external branch on the AP and the `bfq` on the EP. Forcing the comparison to be on the AP may, however, significantly impact code balance between the AP and EP. For example, functions which approximate solutions to equations often iterate until two successive approximations are within a pre-specified limit. In this case the EP is the natural choice to perform the value computations, and moving these computations to the AP would leave the EP with little or nothing to do, negatively impacting performance. Therefore, the compiler takes into account both existing processor markings and the cost of the sub-expression which performs the comparison when determining the branch assignments.

[3] If an AP to EP copy is required then the only consideration is code expansion due to the copy operations. In this case replication of the CSE is employed only if the CSE consists of a small number of low-latency instructions.

Step 5: Propagate All Markings

At this point in the process the entire graph has been traversed and node markings have been propagated to the ancestors and descendants of the anchor nodes. The final step is to traverse the graph once more with node markings being propagated from all nodes. As described previously, copies are inserted where necessary and common sub-expressions may be replicated. The propagation continues as long as it results in changes to the graph (by copy insertion, processor markings, or common sub-expression replication). Once this process terminates there are no unmarked nodes and the two graphs have been extracted.

6 Example Compiler Output

This section presents the compiler output for a simple source file which computes the sum of a sequence of integers (shown in left-most column of Figure 2). This program was chosen because it includes examples of decoupled loads and stores, self loads and stores, copies, external branches, and branch from queue operations. Figure 2 also shows the decoupled access/execute code generated by the compiler. The labels of the form `@FSxx` are stack frame sizes and are simply used to make adjustments to the stack pointer (`r30`). `r1` is the return value register. A single parameter register was used in order to illustrate parameter passing both in a register and on the stack. Relevant assembly language explanations are given in Table 1.

`Opcode`	Description
`mvfq dest, queue`	Move from queue: $dest$ = value at head of $queue$
`brnz src1, label`	Branch not zero: $if(src1)\{PC = label\}$
`brxnz src1, label`	Branch external not zero: $if(src1)\{PC = label\}$, send results of branch to branch queue
`bfq label`	Branch from queue: $if(value\ on\ branch\ queue)\{PC = label\}$

Table 1. Selected Assembly Language Opcodes

Looking at Figure 2 one can see an example of a standard decoupled load on line 14 on the AP (AP.14) and line 21 on the EP (EP.21). AP.2 and EP.3 are the two halves of a standard decoupled store. An example of a slip-reducing copy is on lines EP.1 and AP.11; this particular copy is used to send the parameter passed in register 2 on the EP to the AP. Note that the slip reducing effect is mitigated by placing the producer operation <u>before</u> the SDQ accesses (EP.3 - EP.5) which save temporary registers for the EP, and the consumer operation <u>after</u> the corresponding SAQ, ASAQ, and ASDQ accesses (AP.2-AP.10) which assist the EP in saving its temporaries, and save the AP temporaries. EP.21 and EP.22 set up the first parameter to `sum()` in `r2`. AP.32 and EP.20 store the second parameter to `sum()` on the stack. AP.18 is an external branch and EP.10 is the corresponding branch from queue operation.

Source Code

```
int sol;

int sum(int a, int b)
{
   int i, sum = 0;
   for (i = a; i <= b; i++)
       sum += i;
   return sum;
}

void main(void)
{
   int a = 1, b = 100;
   sol = sum(a,b);
   return;
}
```

Access Processor

```
 1 sum:  addc  r30, r30, @FSaf
 2       addc  SAQ, r30, -56
 3       addc  SAQ, r30, -52
 4       addc  SAQ, r30, -48
 5       addc  ASAQ, r30, -44
 6       mvut  ASDQ, r5
 7       addc  ASAQ, r30, -40
 8       mvut  ASDQ, r4
 9       addc  ASAQ, r30, -36
10       mvut  ASDQ, r3
11       mvut  r2, CPQ
12       addc  ALAQ, r30, b-@FSaf
13       mvfq  r3, ALDQ
14       mvut  r4, r2
15       br    @L5
16 @L3:  addc  r4, r4, 1
17 @L5:  cmpw  r23, r3, r4, LT
18       brxnz r23, @L4
19       br    @L3
20 @L4:  addc  LAQ, r30, -56
21       addc  LAQ, r30, -52
22       addc  LAQ, r30, -48
23       addc  ALAQ, r30, -44
24       addc  ALAQ, r30, -40
25       addc  ALAQ, r30, -36
26       mvfq  r5, ALDQ
27       mvfq  r4, ALDQ
28       mvfq  r3, ALDQ
29       subc  r30, r30, @FSaf
30       ret

31 main: addc  r30, r30, @FScm
32       addc  SAQ, r30, 0-16
33       call  sum
34       addc  SAQ, 0, sol
35       subc  r30, r30, @FScm
36       halt
```

Execute Processor

```
 1 sum:  mvut  CPQ, r2
 2       addc  r30, r30, @FSaf
 3       mvut  SDQ, r5
 4       mvut  SDQ, r4
 5       mvut  SDQ, r3
 6       addi  r5, 0, 0
 7       mvut  r4, r2
 8       br    @L5
 9 @L3:  addc  r4, r4, 1
10 @L5:  bfq   @L4
11       addw  r5, r5, r4
12       br    @L3
13 @L4:  mvut  r1, r5
14       mvfq  r5, LDQ
15       mvfq  r4, LDQ
16       mvfq  r3, LDQ
17       subc  r30, r30, @FSaf
18       ret

19 main: addc  r30, r30, @FScm
20       addi  SDQ, 0, 100
21       addi  r3, 0, 1
22       mvut  r2, r3
23       call  sum
24       mvut  SDQ, r1
25       subc  r30, r30, @FScm
26       halt
```

Fig. 2. Source and Assembly Code for Compilation Example

7 Results

The book *Numerical Recipes in C* contains a wide variety of algorithms, eight of which were selected as a representative sample [5]. These benchmarks have between 82 and 163 lines of source code which result in actual instructions in the executable (i.e., no comments or variable declarations are counted).

The decoupled simulator (*Decsim*) is written in C and accepts a simple object code format containing tuples of `<OPCODE OP1 OP2 OP3>`. The simulator models individual processors that are simple, single-issue, in-order execution, RISC style processors. They employ a five-stage pipeline with hardware interlocks and data forwarding. Each processor has 32 registers and assumes a perfect instruction cache. Infinite depth queues are used to make the results independent of any queue resource constraints. The memory latency of 18 cycles for the first word and 2 cycles for each subsequent word in a memory line/block was selected based on current memory technology. Decsim can operate in either a decoupled or uniprocessor mode. The uniprocessor mode does not use queues, instead re-

lying on standard load and store operations to communicate with memory. In the uniprocessor mode a data cache is employed; no data cache is used in the decoupled mode.

Figure 3 shows the speed-up achieved by running the benchmarks on the two-processor decoupled architecture vs. a uniprocessor architecture with an 8K-byte cache[4] at memory latency of 20 cycles. With the 20 cycle memory the average speed-up is only 1.06, with four of the benchmarks actually running slower on decoupled processor. In all four of poor performing benchmarks the AP spends over 50% of its cycles stalled on either an empty branch or copy queue. As a basis for comparison the original fourteen Livermore Loops were compiled and simulated showing considerable speed-up (2.05 on average) corroborating the results from the previous studies of decoupling using these benchmarks [11, 2].

Intuitively, a speed-up of less than 1.0 seems unlikely since the decoupled processor enjoys a 2:1 advantage in raw processor resources. However, poor decoupled performance can occur if the decoupled processor is unable to achieve slip and the AP experiences the full memory latency on each memory access, while good uniprocessor performance can result if the cache hit rate is high. If these two events occur for the same benchmark the decoupled processor would run significantly slower than the uniprocessor. Investigation into the run-time behavior indicates that the poor performance is attributable to control dependencies and copy operations significantly limiting the slip — research is ongoing on techniques to address these issues. Daecomp was used to perform many other experiments, the results of which are available in [6].

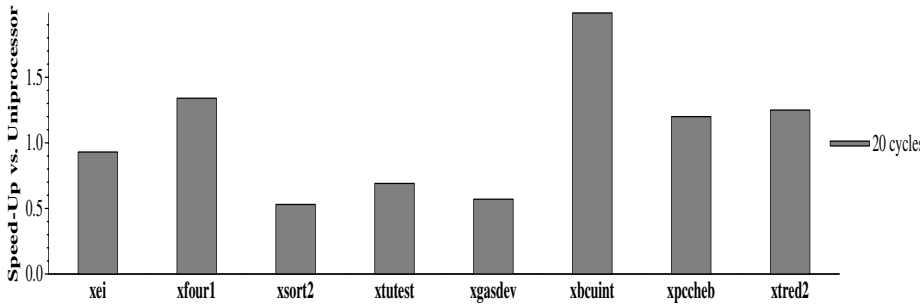


Fig. 3. Speed-Up Comparison with 20 Cycle Memory Latency — NRC

[4] If a typical cache size were used with the benchmarks in question, the cache would like suffer only compulsory/first reference misses, and conflict and capacity misses would likely not be an issue. Therefore, a small cache was used in order to keep it on scale with the working sets of the benchmarks. The cache selected resulted in overall hit rate of 97%.

8 Conclusion

To fully evaluate any architecture a compiler is needed, for this reason the decoupled access/execute compiler Daecomp was constructed. The primary role of a a decoupled compiler is to partition the instructions into two cooperating instruction streams. Daecomp implements a five-step partitioning process to identify the access and execute instruction streams. The results of some of these studies performed with Daecomp confirmed published result obtained using small, hand-compiled benchmarks [11, 1, 4, 2]. However, using larger, more varied benchmarks revealed that many of these earlier conclusions were erroneous, underscoring the importance of constructing a compiler. Further work is planned to determine if the architectural model can be modified to permit speculative execution, or to employ data caches.

References

[1] Ali Berrached, Lee D. Coraor, and Paul T. Hulina. A decoupled access/execute architecture for efficient access of structured data. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 438–447, 1993.

[2] Jian-tu Hsieh, Andrew R. Pleszkun, and James R. Goodman. Performance evaluation of the PIPE computer architecture. Technical Report 566, University of Wisconsin - Madison, November 1984.

[3] G.P. Jones and N.P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[4] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. The effect of memory latency and fine-grain parallelism on Astronautics ZS-1 performance. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, pages 288–296, 1990.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brain P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2 edition, 1996.

[6] Kevin D. Rich. *Compiler Techniques for Evaluating and Extending Decoupled Architectures*. PhD thesis, University of California at Davis, 2000.

[7] James E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[8] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. Compiling and optimizing for decoupled architectures. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995.

[9] N.P. Topham and K. McDougall. Performance of the decoupled ACRI-1 architecture: the Perfect Club. In *Proceedings of High Performance Computing - Europe*, 1995.

[10] Gary S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design*. PhD thesis, University of California at Davis, 1997.

[11] Honesty Cheng Young. Evaluation of a decoupled computer architecture and the design of a vector extension. Technical Report 603, University of Wisconsin-Madison, 1985.

[12] Yinong Zhang and George B. Adams III. Performance modeling and code partitioning for the DS architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

Limits and Graph Structure of Available Instruction-Level Parallelism

Darko Stefanović and Margaret Martonosi

Princeton University, Princeton NJ 08544, USA

Abstract. We reexamine the limits of parallelism available in programs, using run-time reconstruction of program data-flow graphs. While limits of parallelism have been examined in the context of superscalar and VLIW machines, we also wish to study the causes of observed parallelism by examining the structure of the reconstructed data-flow graph. One aspect of structure analysis that we focus on is the isolation of instructions involved only in address calculations. We examine how address calculations present in RISC instruction streams generated by optimizing compilers affect the shape of the data-flow graph and often significantly reduce available parallelism.

1 Background and Related Work

Most studies of the limits of available instruction-level parallelism have focused on the timing of an optimal schedule of the instruction sequence for an idealized processor model. We propose to examine directly the data flow graph of the instruction sequence. Thus we will be able to gain insight into the structural properties of the available parallelism, so that we may understand which elements of the instruction sequence, or which compiler idioms, affect available parallelism. In particular, here we show that the presence of address calculations for memory operations greatly affects parallelism; in some programs, it is precisely the address calculations that limit the asymptotically achievable parallelism. As in earlier studies, we assume no hardware limitations: the degree of parallelism available is the degree exploitable. Examining very long dynamic code sequences means that control flow is entirely revealed and does not constrain parallelism. Imperfect alias analysis in compilers [1, 2] sequentializes code by enforcing the order of the store-load pair, together with all potentially aliased memory operations, for any value that cannot be held in registers and is temporarily stored in memory (register spill, call save, or otherwise); by precise memory disambiguation at run-time we remove all such constraints as well.

A number of studies over the past three decades have looked at the limits of parallelism [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], using instruction-scheduling simulators. The simulator reports the number of cycles needed to execute the program, and the number of instructions executed. The ratio of the two gives the IPC as the standard measure of instruction-level parallelism [11]. The simulator effectively constructs the moving "front line" of the data-flow graph [3]; thus, constructing an entire data-flow graph is not necessary to obtain a single number, the cycle count. However, having an *explicitly* constructed graph permits us to study its structure: we can inspect the computation nodes

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1018–1022, 2000.
© Springer-Verlag Berlin Heidelberg 2000

repeatedly, and evaluate the graph using *multi-pass* and *backward-flow* algorithms. We will illustrate this new possibility on one example: we will recognize instructions involved in address calculations using a backward-flow algorithm.

While in the past reconstructing large graphs was dismissed as impractical [3], that is no longer the case. Currently available memory space permits building graphs sufficiently large to capture interesting application behavior—parallelism analysis using a conceptual dependence graph of a moving window of program execution was demonstrated by Austin and Sohi [1]. Recently, Ebcioğlu *et al.* described a system for dynamic code translation and optimization [12], aimed at transparent porting of applications to a VLIW execution engine. Among other results, they evaluate achieved parallelism without resource constraints, and with store-load bypassing. We obtain comparable parallelism numbers, except for their results with the "combining" optimization, which in some cases show much higher parallelism. This optimization breaks dependence chains of immediate-operand instructions with the dependence on a common register, by adjusting the immediate values (a form of constant folding at the machine level); code modifications are outside the scope of our study.

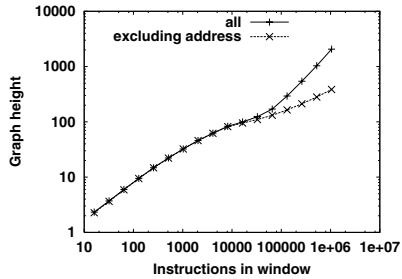
2 Run-Time Analysis of Programs

Our analysis uses the core of the SimpleScalar architectural simulation toolset [13] for the Alpha instruction set, and dynamically constructs a program's data-flow graph. Conceptually, graph nodes correspond to executed instructions, while graph edges correspond to computed operand values. The values are tracked through memory, including multi-byte values through partial and unaligned accesses. This allows us to recognize when entire stored values are reloaded. Nodes are not created for instructions identifiable as data transport: register moves and memory loads; instead, the values are appropriately bypassed from the producing node to the using node. Thus the data flow of the computation is reconstructed independent of the storage layout.

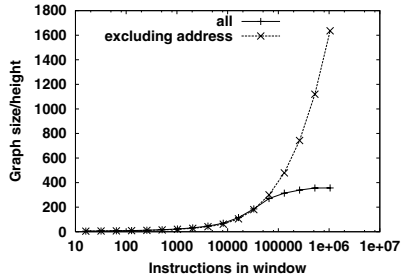
We simulated a number of SPEC95 and Mediabench programs, with up to 1800 million instructions executed. Benchmarks were compiled on a Digital Alpha 21164 EV56 using native C and Fortran compilers, and highly optimized as specified by SPEC. For each benchmark, we varied the size of the instruction window as powers of 2, between 16 and 1M (limited by the memory capacity of the simulator host).

We first look at the parallelism reported for the graphs consisting of all instructions in the examined window; the results are presented in plots (a) and (b) in Figures 1 and 2. The solid lines, labelled *all* in the graph height plots (a), show the growth of average graph height (length of critical path) with increasing instruction window size. The axes in graphs (a) are both logarithmic; the slopes of the curves (below 1) show that the dependence is sublinear. The solid lines, labelled *all* in the graph parallelism plots (b), show the ratio of graph size (number of instruction nodes) to height. This ratio is a measure of average available parallelism, because it reflects the potential speedup of a machine with unbounded hardware resources (limited only by data dependences) over a sequential machine that executes exactly one instruction per cycle in program order. As the instruction window size increases, so does the parallelism. However, we note some distinct behaviors. In *145.fpppp*, the parallelism saturates quickly: with an instruction

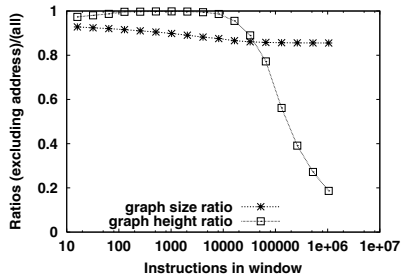
window size of 128K, it is 314, with 1M, it is 357. Not so in *110.applu*: parallelism grows smoothly (but sublinearly) even as very large window sizes are reached. The *absolute* values of parallelism are vastly different: whereas *145.fpppp* achieves over 300, and *110.applu* over 1000, we have only 45 for *129.compress* (not shown). This agrees with observations [3] that some numerical programs have very high intrinsic parallelism, proportional to problem size and exposed by unrolling loops (which we in effect do).



(a) Graph height

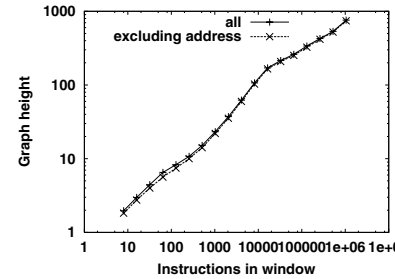


(b) Graph parallelism measure

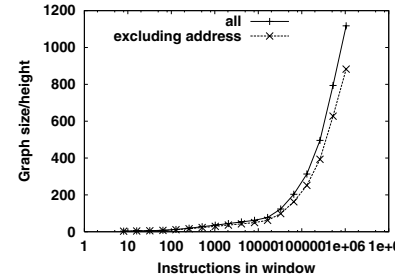


(c) Ratios excluding address calculation

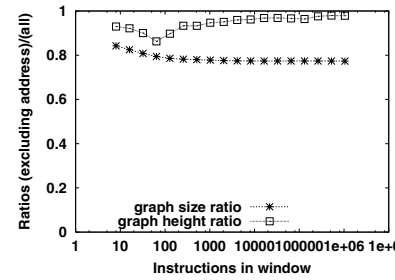
Fig. 1. Benchmark *145.fpppp*



(a) Graph height



(b) Graph parallelism measure



(c) Ratios excluding address calculation

Fig. 2. Benchmark *110.applu*

Excluding Address Calculations. Will there be differences with respect to available parallelism between the data-flow graph as built, and its subgraph that excludes purely address calculations? This is an interesting question, because the latter graph

seems closer to the algorithmic intent of the program, address calculations being partly an artifact of the particular compiler/RISC architecture realization of the program. Recall that while we are reconstructing the data-flow graph at run-time, we are able to recognize when a load instruction L retrieves a value written to memory by a previous store instruction S and produced by a previous computational instruction C. We bypass such a load—an instruction that uses the loaded value sees it instead as coming from C, similar to the *load-store telescoping* optimization [12]. Note that L is no longer needed to represent the computation, and in some cases S also is no longer needed (if L is the only load of the value). Loads and stores are preceded by instructions to calculate an address. (These instructions may in turn include other loads.) If certain loads and stores are no longer needed to represent the computation, then the corresponding address calculations are not needed either. However, while we are building the graph we cannot know which computations will end up being used *only* to calculate addresses. This we determine in a separate, backward-propagating pass over the data-flow graph. (Address calculation recognition subsumes the *stack pointer register analysis* of [10].) The dashed lines, labelled *excluding address* in plots (a) and (b), give the graph height and graph parallelism measure for the data-flow subgraph without address calculations. Plots (c) show the relative size and height of the subgraph with respect to the full graph. We show both in the same plot area to make it easier to compare with the graph parallelism measure plot. (Consider the intersections of (c) curves and the intersections of (b) curves: their abscissæ coincide.) Let us first look at the relative subgraph size, labelled "graph size ratio" in plots (c). This ratio changes very little with instruction window size, and the small observed change is in the direction of somewhat smaller ratios as the window size is increased. Indeed, in the backward-propagating algorithm we must conservatively assume that values present at the end of the instruction window *may* be used as non-addresses in the continuation of the program after the window; as the window grows, the inaccuracy of that assumption diminishes and with it the number of instructions inaccurately assumed to be involved in non-address computation. The ratio varies greatly across benchmarks: 0.9 for *145.fpppp*, 0.8 for *110.applu* and *124.m88ksim*, but just 0.2 for *129.compress*.

Relative subgraph height, labelled "graph height ratio" in plots (c), shows significant variation with window size. In *145.fpppp* it remains close to 1 up to a window size of 16K, but drops sharply thereafter, so that by 1M it is just 0.2. In other words, for smaller windows, the subgraph height is about the same as the full graph height, but for larger windows, the subgraph height collapses. The critical path is determined by a dependence chain of address calculations carried in a loop. If address calculations are eliminated, a much larger amount of parallelism is exposed. We observed the same pattern in *141.apsi*, *099.go*, *134.perl*, *126.gcc*, *130.li*, and *mpeg2decode*. On the other hand, in *110.applu* the ratio of graph heights is close to 1: the critical path is for the most part determined by the "data" calculations, i.e., instructions other than address calculations. We observed a similar pattern in *146.wave5*, *124.m88ksim*, and *adpcm*.

We may summarize the findings as follows: When address calculations form long dependence chains, they can dominate "data" computations, and their removal is beneficial for parallelism. When address calculations are localized, their removal does not affect graph height, yet it reduces graph size; therefore, parallelism is reduced.

3 Future Directions

With data-flow graphs explicitly constructed we are not restricted to critical paths through the entire graph, but can zoom in on particular nodes. For instance, we can examine the critical path of the computation that produces the address for a load (with a view to prefetching), or the critical path that produces a conditional value (with a view to scheduling beyond the corresponding branch). We should consider what can be done in language implementation to reform the way memory data are accessed: a compiler optimization such as array index "strength reduction" can introduce a chain of address calculations where none is apparent at the source level. On the other hand, to appreciate the *practical* repercussions of available parallelism, we should consider code mappings to realistic processors, where memory bandwidth and control flow uncertainty are taken into account. We intend to combine the analysis of instruction-level parallelism with analysis of bit usage [14], which will lead to a finer-granularity description of parallelism as the basis for code mapping decisions for hybrid fixed-configurable processors.

References

[1] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th ISCA*, pages 342–351, May 1992.
[2] J. W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO-28*, Dec. 1995.
[3] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput.*, C-33(11):968–976, Nov. 1984.
[4] D. W. Wall. Limits of instruction-level parallelism. WRL Research Report 93/6, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, CA, Nov. 1993.
[5] C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comput.*, C-21(12):1411–1415, Dec. 1972.
[6] N. P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Trans. Comput.*, 38(12):1645–1658, Dec. 1989.
[7] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *ASPLOS III*, pages 290–302, Boston, Massachusetts, 1989.
[8] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *18th ISCA*, pages 276–286, May 1991.
[9] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th ISCA*, pages 46–57, May 1992.
[10] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. The limits of instructions level parallelism in SPEC95 applications. In *3rd Workshop on Interaction Between Compilers and Computer Architecture*, Oct. 1998.
[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1996. Second Edition.
[12] K. Ebcioğlu, E. R. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *MICRO-32*, Nov. 1999.
[13] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, June 1997.
[14] D. Stefanović and M. Martonosi. On availability of bit-narrow operations in general-purpose applications. In *10th FPL*, Villach, Austria, 2000.

Pseudo-vectorizing Compiler for the SR8000

Hiroyasu Nishiyama, Keiko Motokawa, Ichiro Kyushima, and Sumio Kikuchi

Systems Development Laboratory, HITACHI, Co.Ltd.
{nisiyama,motokawa,kyushima,kikuchi}@sdl.hitachi.co.jp

Abstract. The pseudo-vector processing (PVP) is a framework that enables fast processing similar to vector processing. In this paper, we describe the compiler optimizations that effectively utilize PVP on the SR8000. These include *access method analysis*, *preloading*, and *prefetching* optimizations. Evaluations on the SR8000 indicate that PVP can effectively hides memory latency.

1 Introduction

The SR8000[1, 6] super technical server consists of multiple SMP (Symmetric Multi Processor) nodes. The SMP nodes are connected by high-speed interconnects to provide fast inter-node communications. Each node contains nine RISC microprocessors and main memory. Eight of the microprocessors are used for computation (IP), and the remaining one is used for systems operation (SP).

The microprocessor used for IP and SP is based on the PowerPC[1] architecture. It incorporates a pseudo-vector processing (PVP) mechanism. PVP is a framework that enables fast computation similar to vector processing. It is implemented using the following mechanisms: (1) large floating-point registers and cache memory, (2) a continuous data supply from main memory to the floating-point registers and cache memory using preloading and prefetching, and (3) instruction level parallel execution using pipelining and out-of-order execution.

The following is a list of the special features of the SR8000 related to PVP.

Data prefetch instruction The SR8000 has a 128k byte 4-way set associative data cache with lines of 128 bytes. Each IP can handle up to 16 data prefetch[2] requests simultaneously.

Data preload instruction The data preload instruction loads floating-point data from main memory to a floating-point register directly bypassing cache memory. It can transfer up to 128 floating-point items simultaneously. It does not require useless data transfer even for non-continuous data references.

Extended registers In addition to the 32 floating-point registers defined in the PowerPC specification, 128 extended registers are defined. The section from FR0 to FR31 is called the *global part*, and the section from FR32 to FR128 is called the *slide part*. Floating-point operations are also extended to allow use of the slide part. The slide part can be renamed by software control using

[1] PowerPC is a trademark of IBM Corp.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1023–1027, 2000.
© Springer-Verlag Berlin Heidelberg 2000

slide instructions[5, 3]. After executing a slide instruction, the contents of register $\mathtt{FR}_n$ become accessible by $\mathtt{FR}_m$, where $m = (n-32+P)\ mod\ 128+32$. P means the distance of register number before and after execution of the slide instruction, and is called *slide pitch*.

2 Pseudo-vector Optimization

2.1 Access Method Analysis

We call a method to tolerate memory latency an *access method*. The SR8000 has three access methods: PREFETCH, PRELOAD, and LOAD[2]. The access method for a reference is selected using *access method analysis*.

To select an access method, references in a given loop that have a constant difference of address expression are grouped. An access method is then calculated for each reference group. The access method for a reference is determined by considering the cache line reuse ratio, δ, in a reference group. If δ is larger than or equals to the threshold value, the references of the group can be considerd continuous. If this is the case, PREFETCH is used as the access method of the group; otherwise PRELOAD is used. Since the SR8000 does not support preloading of integer data, PREFETCH is used for integer data even for small δ.

For a group that uses PREFETCH, redundant prefetch instructions can be eliminated using spatial locality. To effectively use spatial reuse, we set the access method for the front reference in the group to PREFETCH as a representative, and set the remaining references to LOAD. The reference length of the group is calculated to insert prefetch instructions for the group, which is described later.

Although the access method for a reference is selected by analyzing its access pattern, static analysis is not always correct. Thus, we also provide user directives and compiler options to explicitly specify access methods.

2.2 Preloading Optimization

Preloading optimization hides memory latency using software pipelining. Kernel creation and slide register allocation are performed assuming memory latency (L) and initiation interval (II). The kernel creation is based on Iterative Modulo Scheduling[4], which iterates kernel creation increasing values of II from MinII, defined by resource and recurrence constraints.

After obtaining a kernel schedule, register allocation for the slide part is performed. Slide register allocation uses an extension of graph coloring algorithm[3]. (1) The slide register allocator obtains live ranges of each floating-point variables. Live ranges that cover whole loop iterations are assigned to the global part. (2) Each candidate live range on the slide part is divided into segments that belong different stage of kernel. We call a group of divided live ranges a *slide group*. (3) An interference graph is created for the divided live ranges. We use graph coloring to assign registers, considering the following restriction: the register number

[2] The case when cache hit is expected.

of node G_i is defined by $(n - 32 + P \times i)\ mod\ 128 + 32$, where the i-th node in a group is G_i, slide pitch is P, and the assigned register number of G_0 is n.

The difference between the preloading optimization from previous studies[4, 3] is that the our compiler retries scheduling after decreasing the estimated value for memory latency, L, when the slide register allocation fails. When the value of L becomes lower than threshold value, latency can not be hidden effectively by preloading. Thus, the access method of the front reference in the group is changed to PREFETCH and the access methods of other references are changed to LOAD making the references candidates for the prefetching.

2.3 Prefetching Optimization

Prefetching optimization hides memory latency by inserting prefetch instructions for references with PREFETCH access method. For each reference group, the number of iterations, N, required to hide latency is defined as $\lceil$Memory Latency/Estimated Cycle of Loop$\rceil$. Thus the front address of prefetching for a group G is defined as $F(G) + N \times S(G)$, where $F(G)$ is the front reference address of G, and $S(G)$ is the distance in the addresses referenced by G between iterations *i-th* and $i+1$-*th*. To prefetch every item in the group, $\lceil Len[G]/128 \rceil$ prefetch instructions are issued from the front address in every cache line, where $Len[G]$ is reference length of G. Since the front reference may not be on the cache line border, a prefetch instruction should be issued for the last reference when the reference length is not a multiple of cache line size. In addition, when difference between the next to the last prefetching address, $F(G_i) + \lfloor Len[G_i]/128 \rfloor \times 128$, for the *i-th* iteration of G and the first reference address, $F(G_{i+1})$, for the $i+1$-*th* iteration of G is smaller than the cache line size, we can assume that the reference of G is consecutive between iterations. Thus the prefetching of the last reference in the group is not required in this case.

Since the cache line of the SR8000 is relatively large (128 bytes), prefetching for small stride reference is a waste of instruction bandwidth. The SR8000 compiler eliminates redundant instructions by unrolling the loop without eliminating exit branches to reduce the overhead of short loops.

3 Evaluation

We describe the results of evaluations on the SR8000. All evaluations were performed on a single node of the SR8000 without automatic SMP parallelization using a hardware performance counter so as to precisely evaluate the effect of PVP. In the following discussion, we denote the case that uses neither prefetching nor preloading by LD, the case that uses only preloading by PLD, the case that uses only prefetching by PF, and the case that selectively uses prefetching and preloading via automatic analysis by PLD+PF.

Basic loop performance Figure 1(a) shows the performance of a DAXPY (`Y[i]=A*X[i]+Y[i]`) loop for a range of stride values. The performance of LD was low even for sequential access. The performance of PLD was stable and

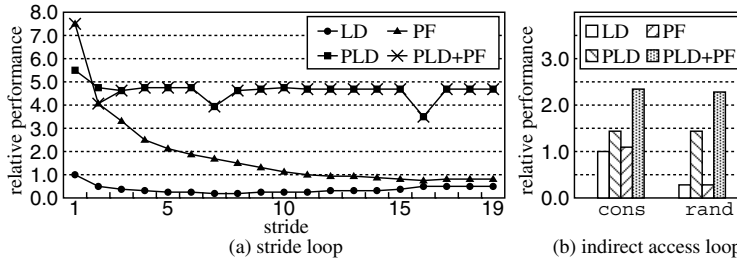


Fig. 1. Basic loop performance

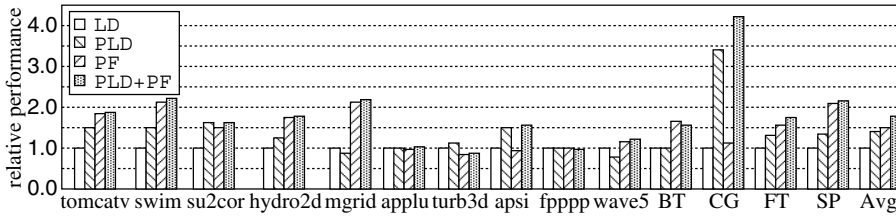


Fig. 2. Performance of SPEC&NPB

4∼5 times better than sequential LD case. PF achieved still higher performance for sequential access. However the performance dropped rapidly to the level of LD as stride value increased. Automatic analysis, assuming that the threshold value of δ is 0.5, showed the most stable performance. This method selects an appropriate method for each stride value, except for a stride value of 2.

Figure 1(b) shows the results of an indirect access loop. The indirect access loop (`Z[L[i]]=A*X[L[i]]+ Y[L[i]]`) was tested for having consequtive (**cons**) and random (**rand**) elements. The performance is shown relative to that of LD with consequtive elements. For the case where the values of `L` were consecutive, PF obtained slightly better performance than LD because of cache reuse. However, the performance gain is small since there is instruction overhead for each loop iteration for prefetching. PLD showed much larger performance improvements by hiding the memory latency with low instruction overhead even without hiding the memory latency of `L`. Hiding memory latency with a combination of preloading and prefetching showed better improvements than other methods because memory latency is effectively hidden. The performance of PF and LD drops remarkably for the random access case. However PLD and PLD+PF also hold better performance than LD. This shows good toleration of memory latency by pseudo-vector processing using access method analysis.

SPEC&NPB benchmarks Figure 2 shows the results of performance evaluation on SPEC95fp benchmarks and four benchmarks (BT,CG,FT,SP) from

NAS Parallel Benchmarks. The performances of PLD, PF, and PLD+PF relative to the performance of LD are shown in this figure. We used 'ref' data for SPEC and class B for NAS.

The maximum performance gain of PLD is 237% on CG(avg. 34%), the maximum performance gain of PF is 114% on swim(avg. 45%), and the maximum performance gain of PLD+PF is 323% on CG(avg. 71%). Automatic selection shows equivalent or better performance than use of only a single latency hiding method for all benchmarks except turb3d. For a preloading loop, when the loop iteration count is small, it is difficult to issue preload instructions in advance using software pipelining. Further, when the dependencies between array references are uncertain at compile time, preload instructions are also difficult to issue in advance. These are the reasons for the low performance of PLD on benchmarks such as mgrid. On the other hand, the preloading, for benchmarks such as CG, outperforms prefetching. The reason is better tolerance of non-continuous data references and prefetching's performance degradation from cache slashing. Since data preloading bypasses cache access, a performance drop from cache slashing does not occur.

4 Conclusion

The SR8000 can hide memory latency effectively with software managed controlling of data movement using pseudo-vector processing. In this paper, we have described optimizations of the pseudo-vectorizing compiler for the SR8000. According to the results of this evaluation, the SR8000 shows good tolerance of memory latency with pseudo-vectorization.

References

[1] T. Kurihara, K. Shimada, E. Kamada, and T. Shimizu. A RISC Processor for SR8000: Accelerating Large Scale Scientific Computing with SMP. In *Hot Chips 11*, 1999.

[2] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, 1992.

[3] K. Nakazawa, H. Nakamura, H. Imori, and S. Kawabe. Pseudo Vector Processor Based on Register-Windowed Superscalar Pipeline. In *Proceedings of Supercomputing'92*, pages 642–651, 1992.

[4] B.R. Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *Proceedings of the 27th Symposium on Microarchitecture*, pages 63–74, 1994.

[5] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 Departmental Supercomputer: Design philosophies, decisions and trade-offs. *Computer*, 22(1):12–35, 1989.

[6] Y. Tamaki, N. Sukegawa, M. Ito, Y. Tanaka, M. Fukagawa, T. Sumimoto, and N. Ioki. Node Architecture and Performance Evaluation of the Hitachi Super Technical Server SR8000. In *Int'l Conference on Parallel and Distributed Computing and Systems*, pages pp.487–493, 1999.

Topic 15
Object Oriented Architectures, Tools, and Applications

Gul A. Agha

Global Chair

For some time, object-oriented programming has become standard practice in sequential programming. Objects separate the interface from the representation and promote reuse of code. Although concurrency is a natural consequence of objects, the standard model of objects uses sequential procedure calls. Early research in actors unified concurrency with objects and provided a basis for use of objects in parallel and distributed systems.

The Euro-Par conference, as its predecessor PARLE, has a long tradition of covering cutting edge research in concurrent object-oriented programming. Since the late 1980s, the field has matured, and increasingly, research is being conducted on architectures, tools and applications. Our call for papers for this workshop chose to emphasize this aspect. Part of this shift has been the widespread acceptance of Java which incorporates some support for concurrency and distribution.

Five regular papers and one short paper were chosen for this workshop from twice as many submissions. The three papers in the first session are closely tied to Java.

The first paper in the workshop by Ngo and Barton discusses how *reflection* may be employed across distributed platforms on a Java Virtual Machine written in Java. In Java, the term reflection refers to the ability of objects to describe themselves (also called reification). This paper proposes to provide reflection remotely so that code can be inspected and debugged during execution. One of the unfortunate features of Java is the fact that it does not provide a uniform address space. Objects are referenced with respect to their current location and their reflection code must be executed within the same address space as that in which the objects reside. Ngo and Barton show how to address this difficulty.

The paper by Antoniu et al. focuses on compilation of Java bytecode into native code in a distributed memory environment. The idea is to provide transparent distribution by executing a single Java virtual machine over a shared memory abstraction. The idea that code does not have to be rewritten for a different architecture is always an attractive one. Experience will show the eventual applicability of such an approach. For example, location awareness rather than transparency may facilitate load balancing. However, the performance results reported in this paper show the approach is promising.

In the final paper in the Java related session, Chiao, Wu and Yuan describe an alternative to Java's concurrency constructs called EMonitor. One of the difficulties in using Java for concurrent programming is that a programmer is forced

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1029–1030, 2000.
© Springer-Verlag Berlin Heidelberg 2000

to do low-level synchronization. Objects are data encapsulation boundaries but not concurrent execution boundaries; instead they serve as surrogates for thread synchronization. Java provides simple, efficient mechanisms for this, allowing methods in objects to be explicitly locked to protect against undesirable concurrent access. This solution suffers from a number of problems such as single condition queues, deadlocks of inter-monitor nested calls, etc. The paper discusses these problems and offers EMonitor as an alternate which provides more flexible multithreaded programming under high-contention without significant performance overhead in the low-contention case.

The second session starts with a paper by Tran and Gérodolle on an object-oriented framework for building "large-scale real-time networked virtual environment applications." Their system has been used for multi-player game prototypes. The paper illustrates an increasingly accepted trend in software – using middleware to effectively separate policy from mechanism. Incidentally, the system described has also been implementated in Java.

The paper by Nolte, Sato and Ishikawa describes a template library for data parallel programming on distributed objects. The idea is to exploit the polymorphism in function templates mechanism of C++ and provide reusable topology classes (such as grips, lists, and trees). The topologies can be used for globally synchronized operations. While performance comparisons to collective operations of MPI are in a competitive range, it would be interesting to see how the performance compares with asynchronous versions of algorithms that make use of the overlap of computation and communication to improve execution efficiency.

The final paper of the session by Grundman, Ritt and Rosenstiel concisely describes a message-passing library implemented using C++. It provides type-safety and easy transmission of objects. The library is an effective way of systematically extending the messaging facilities using object-oriented techniques.

Debugging by Remote Reflection

Ton Ngo and John Barton*

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
ton@us.ibm.com, john_barton@hpl.hp.com

Abstract. Reflection in an object-oriented system allows the structure of objects and classes to be queried at run-time, thus enabling "meta-object" programming such as program debugging. Remote Reflection allows objects in one address space to reflect upon objects in a different address space. Used with a debugger, remote reflection makes available the full power of object-oriented reflection even when the object examined is within a malfunctioning or terminated system. We implemented remote reflection as an extension to an interpreter to create a very effective debugger for Jalapeño, a Java Virtual Machine written in Java.

1 Introduction

Reflection in an object-oriented language supports programs that manipulate the fields (data values) of an object using symbolic names specified at run-time. For instance, an object *obj* may provide a method *getClass()* to describe its own type, the type (class) in turn may provide a method *getFields()* to describe its fields (data values), and the field object may provide a *get()* method for accessing the corresponding data value from an object.

Reflection enjoys extensive support in several modern object-oriented languages such as Java. A standard Java object provides numerous reflective methods for querying its internal values. The package *java.lang.reflect* provides a complete set of utilities to manage Java objects reflectively. With the description of the object encoded in the reflection methods, a program can inspect and manage an arbitrary object without any special knowledge about the object. This meta-object programming[5, 9] is especially useful for system components or utility programs such as debuggers.

In an object-oriented system, reflective methods are encapsulated within the object; therefore to access the internal values of an object, the reflection code must be executed in the same address space where the data resides. Although this is the desired behavior in most cases, debugging is one case where this encapsulation of code and data may present a problem. The reason is that a program being debugged generally needs to be halted, i.e. its execution frozen at an arbitrary point, so that its values and states can be inspected reliably.

* Current address: Hewlett Packard Laboratories, MS 1U-17, 1501 Page Mill Road, Palo Alto, CA 94304

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1031–1038, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In the case of debugging user applications, the debugger can still take full advantage of reflection since the user application is running on a stable system. The system can halt the application thread but continue to execute the debugger thread. Such a debugger is typically called *in-process* because it runs in the same process as the program being debugged.

For the case of debugging system code, reflection is not possible for several reasons. First, halting the system itself would prevent the system from responding to any reflective queries. Second, allowing the system to execute the request will unintentionally change the states of the system while it is being inspected[1]. Third, if the system crashes, the core image can be saved and inspected post-mortem, but no code can be executed. To solve these problems, the debugger must execute in a different process and control the system being developed through some debugging interface provided by the operating system. Consequently, such a debugger is called *out-of-process*.

This situation arose in the development of Jalapeño[1, 2], a virtual machine for Java servers under development at the IBM T. J. Watson Research Center. Jalapeño is a compile-only system: instead of being interpreted, a method is compiled and optimized directly into machine instructions. Because the entire system is written in Java including the runtime, the compiler and garbage collector, reflection is used extensively to integrate the various components; consequently, there is a strong motivation for the Jalapeño debugger to use the same reflection facilities to inspect the system.

In this paper, we present *Remote Reflection* as a technique that allows a program to execute a reflection method on an object that resides in a different process. In our case, this technique allows the debugger to make reflective queries to the Jalapeño system that has been completely halted in a different process. Remote reflection thus extends the power of reflection across different address spaces, improving the reusability of object-oriented codes. Although this technique was developed for Jalapeño and Java, we believe it is applicable to other Java implementations and other object-oriented languages.

In the remainder of the paper, we will discuss remote reflection within the context of Java. Section 2 describes the general programming model for using remote reflection. Section 3 describes an implementation of remote reflection for Jalapeño. Section 4 illustrates the implementation with a detailed example, and possible further developments are discussed in Section 5.

1.1 Related Works

The Sun JDK debugger [8] and the more recent Java Platform Debugger Architecture [7] are also out-of-process and are based on reflection; however, there are several important differences. First, the Sun approach requires a debugging thread running internally in the virtual machine, dedicated to responding to external queries. For Jalapeño this is not possible for the reasons described earlier.

[1] Consider debugging thread scheduling code: dispatching the debugger thread itself would change the thread states

Second, the reflection interface for the debugger is different from the internal reflection interface. In contrast, remote reflection requires no effort on the target system and the same reflection interface is used internally or externally.

2 Remote Reflection

Consider (1) a Java Virtual Machine (JVM) in a remote process that has been halted at an arbitrary point; (2) a program written based on the reflection interface of this remote JVM; and (3) another JVM in a local process executing this program.

Remote reflection allows the program in the local JVM to execute a reflection method that operates directly on an object residing in a remote JVM. The key to remote reflection is a proxy object in the local JVM called the *Remote Object* which represents the real object in the remote JVM.

As illustrated in Figure 1, the programming model for remote reflection is simple yet effective. The user specifies that certain methods of the reflection interface will return remote objects from a different JVM. These methods in the local JVM are said to be *mapped* to the remote JVM since they serve as the link between the two JVM's. Once a remote object is obtained from a mapped method, all values or objects derived from it will also originate from the remote JVM. Aside from the list of mapped methods, a remote object is indistinguishable from a normal object in the local JVM from the program perspective.

Consider a simple example in Figure 2. To compute the line number, the method *Debugger.lineNumberOf()* obtains a table of *VM_Method*'s, selects the desired element and invokes its virtual *getLineNumberAt()* method. This reflection method then consults the object's internal array to return a line number. Supposed that on a local JVM with remote reflection, the static method *VM_Dictionary.getMethods()* has been mapped to an array of *VM_Method*'s in the remote space. When we execute *lineNumberOf()*, the variable *methodTable* receives the initial remote object from *VM_Dictionary.getMethods()*. The variable *candidate* then gets another remote object from accessing the remote array, and finally the method *getLineNumberAt()* is invoked on the remote object.

The uniform treatment of local and remote objects provides the main advantage of remote reflection. Because a remote object is logically identical to a local object, a program uses the same reflection interface whether it executes in-process or out-of-process. As a result, the maintenance of both the reflection interface and programs using it is greatly simplified.

A second advantage is that no effort is required in the remote JVM, since remote reflection relies on the underlying operating system to access the JVM address space.

Finally, mapping per method instead of per class allows flexibility in selecting the object to be mapped. A class may have some instances in the local process and other instances in the remote process without conflict. While a mapped method is not necessarily tied to a single object in the remote JVM, in practice, it is more convenient to map *access* method which return a specific object.

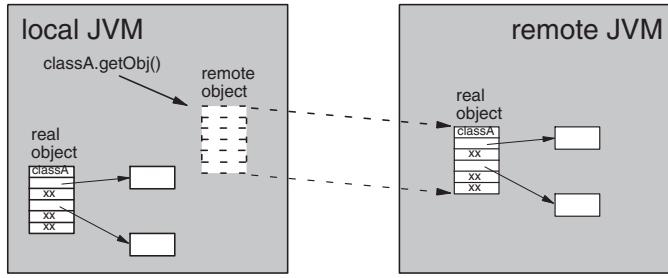


Fig. 1. Programming model for remote reflection: certain methods, e.g. *classA.getObj()*, are specially designated to return *remote objects* that are proxies for the real objects in the remote JVM. In this Figure, the boxes in each JVM represent objects with fields. There are two real instances of classA: one in the local JVM and one in the remote JVM.

3 Implementation

In this section, we describe an implementation in the Jalapeño system. In Java, remote reflection is supported at the level of the virtual machine by either the interpreter or the runtime compiler. Our debugging environment involves three components: the Jalapeño system being debugged, the debugger, and a Java interpreter that has been extended to support remote reflection.

The extension includes managing the remote object and extending the bytecodes to operate on the remote object. Remote reflection also requires operating system support for access across processes. This functionality is typically provided by the system debugging interface, which in the Jalapeño implementation is the Unix *ptrace* facility. Our implementation is simplified by the fact that the debugger only makes queries and does not modify the remote JVM (except explicitly by a user command); therefore, we do not have to address the issue of creating new objects in the remote space.

3.1 Remote Object

The remote object is simply a wrapper that holds sufficient information to find the real object in the remote process. For Jalapeño, this includes the type of the object and its real address.

Remote objects originate from a mapped method or another remote object. In first case, the address is provided to the interpreter from the process of building the boot image[1]. For the latter case, the address is computed based on the field offset from the address of the remote object.

For native methods, a complete implementation will involve extending the JNI implementation to handle remote objects. However in our implementation, it was sufficient to clone the remote object and remote 1-D array of primitives because this satisfies the need of the debugger.

3.2 Bytecode Extensions

Since the initial remote object is obtained via a *mapped* method, the bytecode *invokestatic* or *invokevirtual* to invoke a method are extended as follows. The target class and method are checked against the mapping list. Those to be mapped are intercepted so that the actual invocation is not made. Instead, if the return type is an object, a remote object is created containing the type and the address of the corresponding object in the remote JVM. If the return type is a primitive, the actual value is fetched from the remote JVM.

In addition, all bytecodes that operate on a reference need to be extended to handle remote objects appropriately - for Java, this includes 23 bytecodes. If the result of the bytecode is a primitive value, the interpreter computes the actual address, makes the system call to obtain the value from the remote address space, and pushes the value onto the local Java stack. If the result is an object, the interpreter computes the address of the field holding the reference, makes the system call to obtain the field value and pushes onto the Java stack a new remote object with the appropriate type.

4 Example

In this section, we return to the example in Figure 2 to analyze the actions that occur when the call *lineNumberOf(5,4)* is executed. For reference, Figure 2 also shows the bytecodes for the methods with dashed lines correlating the Java source lines with its bytecodes.

In Figure 3, the box at the right represents the remote JVM, showing a number of objects that have been created in its space. Recall that the static method *VM_Dictionary.getMethods()* has been mapped to the array of *VM_Method* objects in the remote space. The states of the Java stack at successive points are shown in the top and bottom rows, labeled with highlighted numbers from 1 to 11. The state numbers are cross-referenced between Figure 2 and Figure 3. Also shown in Figure 3 are the remote objects (center) in the local JVM that serve as proxies to the corresponding real objects in the remote JVM. Due to limited space, we will only examine in details states 1-3 at the beginning and states 9-11 at the end; the remaining states exhibit similar behavior.

First, the interpreter recognizes *VM_Dictionary.getMethods()* as a mapped method and intercepts the bytecode *invokestatic* to create the initial remote object. The remote object contains the return type *VM_Method* array and the address of the real array in the remote process. In Figure 3, the Java stack in state (1) shows the local variable *methods* on top of the stack and holding a reference to the newly created remote object.

The following bytecodes *astore_3*, *aload_3* and *iload_1* make the preparation for accessing the remote array, resulting in Java stack state (2). When the interpreter executes the bytecode *aaload* to access an array element, it detects that the array reference is a remote object. Since it is an array of objects, the interpreter determines the element type, computes the address and pushes onto the stack the new remote object, resulting in Java stack state (3).

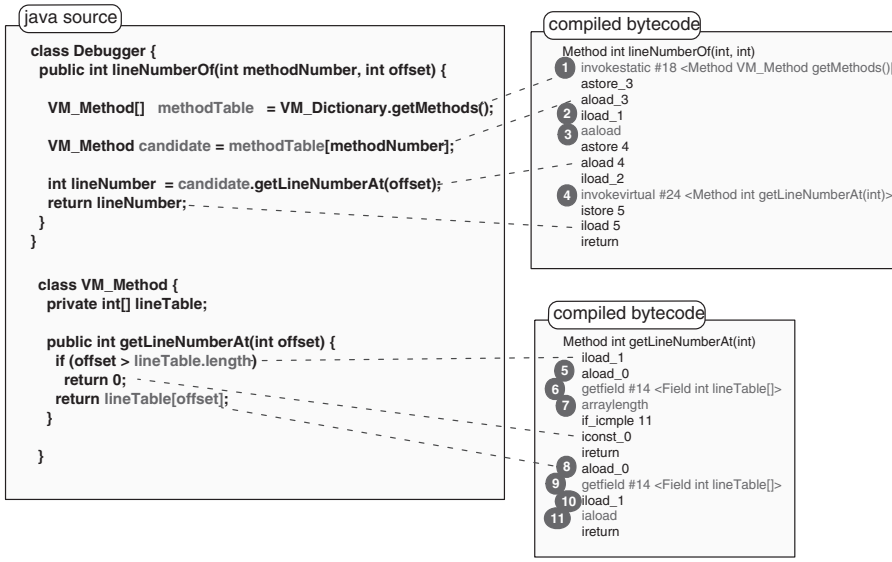


Fig. 2. Example: Java programs making reflective queries and the corresponding bytecodes. The dashed lines correlates the Java source lines with its bytecodes. The highlighted numbers refer to successive states of the Java stack during the program execution; they are cross-referenced with Figure 3.

The execution continues likewise with more remote objects created. To arrive at state (9), the bytecode *getfield* accesses the field *lineTable* and the interpreter creates another remote object on top of the stack. In state (10), the array index is pushed onto the stack and the interpreter executes *iaload* to access the array element. It detects that the array is remote and that the element type is an integer. The interpreter computes the address of the array element and makes the system call to read the value from the remote JVM space. In state (11), the value from the remote array is placed on the stack as a return value.

It is worth noting that remote objects are always temporary; they only exist on the Java stack because they contain real addresses in the remote process that are only valid until the remote JVM resumes execution.

5 Status and Future Works

The interpreter with the remote reflection extension was completed and together with the Jalapeño debugger has been indispensable in the development of the Jalapeño system. Future extensions include several possibilities.

For a production JVM system, debugging requires additional care since the system cannot be taken down and restarted for each debugging session. In this situation, remote reflection must be able to connect to the running system with-

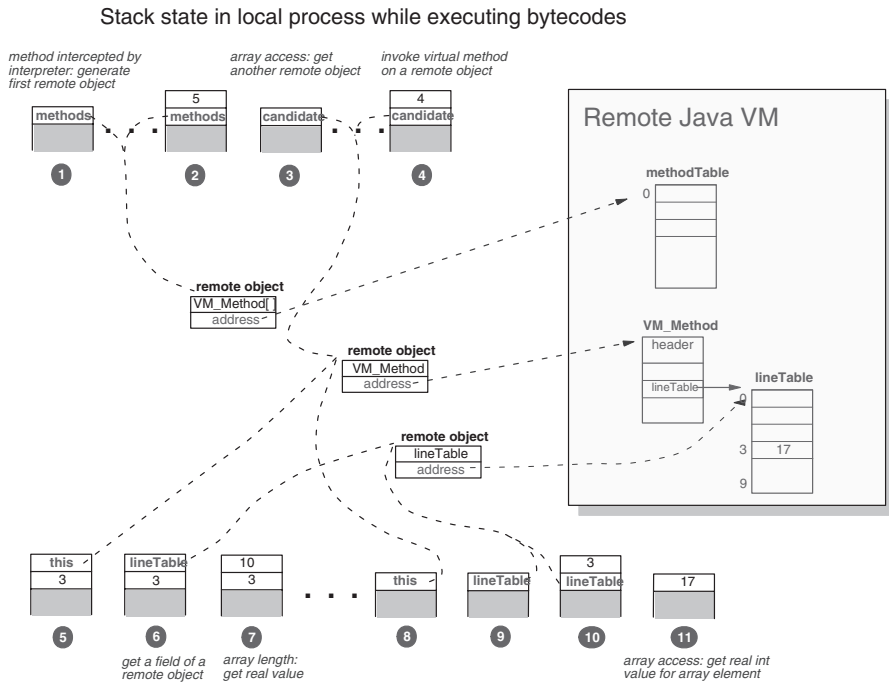


Fig. 3. Example: the successive states of the Java stack (top and bottom rows) during the execution of the reflective queries, showing the remote objects being computed and resolved. The states are labeled with highlighted numbers to cross-reference with the bytecodes in Figure 2. The *Remote Java VM* box (right) shows three real objects existing in the remote space, while the boxes in the center are the remote objects serving as proxies for the real objects.

out any significant side effect. A system facility other than *ptrace* will be necessary so that the running system would retain most of its own process control.

Remote reflection can also be useful in the Java Platform Debugger Architecture of Java 2. We would only need to base the Java Debug Interface implementation directly on the internal reflection interface of the target JVM. This JDI implementation and a JDI-based debugger would run on another JVM that has been extended for remote reflection. This configuration would then bring to JPDA the same capabilities in debugging Jalapeño: low level debugging and the ability to halt the JVM to avoid perturbing its state.

For distributed Java applications runing on several remote JVM's, remote reflection can provide the convenience of a shared memory programming model, allowing them to readily access remote objects. However, since the applications would not be halted, synchronization as well as other issues will need to be studied carefully.

6 Conclusions

Reflection is an important addition to object-oriented systems. In this paper, we describe remote reflection, a transparent mapping technique that preserves the benefits of reflection in situations where it is necessary to decouple the code and the data involved in reflection because they reside in different address spaces.

While the concept of reflection programming has been used across processes, our technique offers several advantages not present in previous efforts. First, it is not necessary to define a different reflection interface to use across processes; the same interface is used whether the program is in-process or out-of-process. Second, no effort is required in the target process; therefore it does not need to be functional and its state is not perturbed.

We describe a simple programming model using remote reflection and an implementation in the Jalapeño system, a Java Virtual Machine developed at the IBM T. J. Watson Research Center. In this context, remote reflection is used to support an out-of-process debugger for the Jalapeño system code. Remote reflection allows the debugger to exploit the benefits of both in-process and out-of-process debugging, resulting in a very effective tool for developing an object-oriented system.

References

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio J. Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 2000, Vol 39, No 1, pp 211-238.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice Shepherd, and Stephen Smith. Implementing Jalapeño in Java. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, November 1999, pp 314-324.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future, The Story of Squeak. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 1997, pp 318-326.

[5] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1992.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 1996.

[7] Sun Microsystems. Java 2 SDK Standard Edition.

[8] Sun Microsystems. Java Development Kit 1.1.

[9] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

Compiling Multithreaded Java Bytecode for Distributed Execution

Gabriel Antoniu[1], Luc Bougé[1], Philip Hatcher[2], Mark MacBeth[2⋆], Keith McGuigan[2], and Raymond Namyst[1]

[1] LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.
Raymond.Namyst@ens-lyon.fr
[2] Dept. Computer Science, Univ. New Hampshire, Durham, NH 03824, USA.
Philip.Hatcher@unh.edu

Abstract. Our work combines Java compilation to native code with a run-time library that executes Java threads in a distributed-memory environment. This allows a Java programmer to view a cluster of processors as executing a *single* Java virtual machine. The separate processors are simply resources for executing Java threads with true concurrency and the run-time system provides the illusion of a shared memory on top of the private memories of the processors. The environment we present is available on top of several UNIX systems and can use a large variety of network protocols thanks to the high portability of its run-time system. To evaluate our approach, we compare serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. All measurements have been carried out on two platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

1 Introduction

The Java programming language is an attractive vehicle for constructing parallel programs to execute on clusters of computers. The Java language design reflects two emerging trends in parallel computing: the widespread acceptance of both a threads programming model and the use of a distributed-shared memory (DSM). While many researchers have endeavored to build Java-based tools for parallel programming, we think most people have failed to appreciate the possibilities inherent in Java's use of threads and a "relaxed" memory model.

There are a large number of parallel Java efforts that connect multiple Java virtual machines by utilizing Java's remote-method-invocation facility (e.g., [4, 5, 10, 13]) or by grafting an existing message-passing library (e.g., [7, 8]) onto Java. In our work we view a cluster as executing a single Java virtual machine. The separate nodes of the cluster are hidden from the programmer and are simply resources for executing Java threads with true concurrency. The separate

⋆ Mark MacBeth is currently affiliated with Sanders, A Lockheed Martin Company, PTP02-D001, P.O. Box 868, Nashua, NH, USA.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1039–1052, 2000.
© Springer-Verlag Berlin Heidelberg 2000

memories of the nodes are also hidden from the programmer and our implementation must support the illusion of a shared memory within the context of the Java memory model, which is "relaxed" in that it does not require sequential consistency.

Our approach is most closely related to efforts to implement Java interpreters on top of a distributed shared memory [2, 6, 17]. However, we are interested in computationally intensive programs that can exploit parallel hardware. We expect the cost of compiling to native code will be recovered many times over in the course of executing such programs. Therefore we focus on combining Java compilation with support for executing Java threads in a distributed-memory environment.

Our work is done in the context of the Hyperion environment for the high-performance execution of Java programs. Hyperion was developed at the University of New Hampshire and comprises a Java-bytecode-to-C translator and a run-time library for the distributed execution of Java threads. Hyperion has been built using the PM2 distributed, multithreaded run-time system from the École Normale Supérieure de Lyon [12]. As well as providing lightweight threads and efficient inter-node communication, PM2 provides a generic distributed-shared-memory layer, DSM-PM2 [1]. Another important advantage of PM2 is its high portability on several UNIX platforms and on a large variety of network protocols (BIP, SCI, VIA, MPI, PVM, TCP). Thanks to this feature, Java programs compiled by Hyperion can be executed with true parallelism in all these environments.

In this paper we describe the overall design of the Hyperion system, the strategy followed for the implementation of Hyperion using PM2, and a preliminary evaluation of Hyperion/PM2 by comparing serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. The evaluation is performed on two different platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

2 The Hyperion System

2.1 Compiling Java

Our vision is that programmers will develop Java programs using the workstations on their desks and then submit the programs for production runs to a "high-performance Java execution server" that appears as a resource on the network. Instead of the conventional Java paradigm of pulling bytecode back to their workstation for execution, programmers will push bytecode to the high-performance server for remote execution. Upon arrival at the server the bytecode is translated for native execution on the processors of the server. We utilize our own Java-bytecode-to-C compiler (`java2c`) for this task and then leverage the native C compiler for the translation to machine code.

As an aside, note that the security issues surrounding "pushing" or "pulling" bytecodes can be handled differently. When pulling bytecodes, users want to

bring applications from potentially untrusted locations on the network. The Java features for bytecode validation can be very useful in this context. In contrast, when "pushing" bytecodes to a high-performance Java server, conventional security methods might be employed, such as only accepting programs from trusted users. However, the Java security features could still be useful if one wanted to support an "open" Java server, accepting programs from untrusted users.

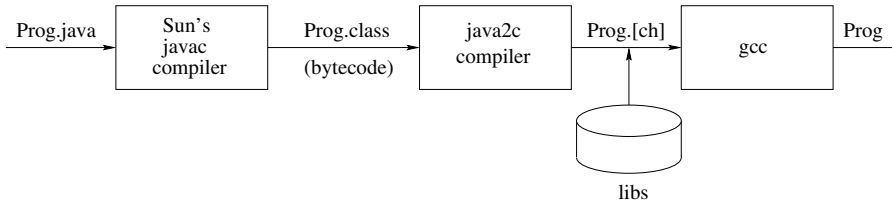


Fig. 1. Compiling Java programs with Hyperion

Code generation in `java2c` is straightforward (see Figure 1). Each virtual machine instruction is translated directly into a separate C statement, similar to the approaches taken in the Harissa [11] or Toba [14] compilers. As a result of this method, we rely on the C compiler to remove all the extraneous temporary variables created along the way. Currently, `java2c` supports all non-wide format instructions as well as exception handling.

The `java2c` compiler also includes an optimizer for improving the performance of object references with respect to the distributed-shared memory. For example, if an object is referenced on each iteration of a loop, the optimizer will lift out of the loop the code for obtaining a locally cached copy of the object. Inside the loop, therefore, the object can be directly accessed with low overhead via a simple pointer. This optimization needs to be supported by both compiler analysis and run-time support to ensure that the local cache will not be flushed for the duration of the loop.

2.2 The Hyperion Run-Time System Design

To build a user program, user class files are compiled (first by Hyperion's `java2c` and then the generated C code by a C compiler) and linked with the Hyperion run-time library and with the necessary external libraries. The Hyperion run-time system is structured as a collection of modules that interact with one another (see Figure 2). We now present the main ones.

Java API Support. Hyperion currently uses the Sun Microsystems JDK 1.1 as the basis for its Java API support. Classes in the Java API that do not include native methods can simply be compiled by `java2c`. However, classes with native methods need to have those native methods written by hand to fit the Hyperion

design. Unfortunately, the Sun JDK 1.1 has a large number of native methods scattered throughout the API classes. To date, we have only implemented a small number of these native methods and therefore our support for the full API is limited. We hope that further releases of Java 2 (e.g., Sun JDK 1.2) will be more amenable to being compiled by `java2c`.

Threads Subsystem. The threads module provides support for lightweight threads, on top of which Java threads can be implemented. This support obviously includes thread creation and thread synchronization. For portability reasons, we model the interface to this subsystem on the core functions provided by POSIX threads. Thread migration is also available, thanks to PM2's support. We plan to use this feature in future investigations of dynamic and transparent application load balancing.

Communication Subsystem. The communication module supports the transmission of messages between the nodes of a cluster. The interface is based upon message handlers being asynchronously invoked on the receiving end. This type of interface is mandatory since most communications, either one-way or round-trip, must occur without any explicit contribution of the remote node: incoming requests are handled by a special daemon thread which runs concurrently with the application threads. For example, in our implementation of the Java memory model, one node of a cluster can asynchronously request data from another node.

Memory Subsystem. The Java memory model [9] allows threads to keep locally cached copies of objects. Consistency is provided by requiring that a thread's object cache be flushed upon entry to a monitor and that local modifications made to cached objects be transmitted to the central memory when a thread exits a monitor. Table 1 provides the key primitives of the Hyperion memory subsystem that are used to provide Java consistency. The DSM environment on top of which they are built is required to provide direct support for their implementation. This condition is fulfilled by the API of the DSM layer of PM2 (see Section 3.2 for additional details).

`loadIntoCache`	Load an object into the cache
`invalidateCache`	Invalidate all entries in the cache
`updateMainMemory`	Update memory with modifications made to objects in the cache
`get`	Retrieve a field from an object previously loaded into the cache
`put`	Modify a field in an object previously loaded into the cache

Table 1. Key DSM primitives

Hyperion's memory module also includes mechanisms for object allocation, garbage collection and distributed synchronization. Java monitors and the associated wait/notify methods are supported by attaching mutexes and condition variables from the Hyperion threads module to the Java objects managed by the Hyperion memory layer.

Load Balancer. The load balancer is responsible for choosing the most appropriate node on which to place a newly created thread. The current strategy is rather simple: threads are assigned to nodes in a round-robin fashion. We use a distributed algorithm, with each node using round-robin placement of its locally created threads, independently of the other nodes. More complex load balancing strategies based on dynamic thread migration and on the interaction between thread migration and the memory consistency mechanisms are currently under development.

3 Hyperion/PM2 Implementation Details

The current implementation of Hyperion is based on the PM2 distributed multithreaded environment (Figure 2). PM2's programming interface allows threads to be created locally and remotely and to communicate through RPCs (Remote Procedure Calls). PM2 also provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such functionality is typically useful to implement dynamic load balancing policies. The interactions between thread migration and data sharing are handled through a *distributed shared memory* facility: the *DSM-PM2* [1] layer.

Most Hyperion run-time primitives in the threads, communication and shared memory subsystems are implemented by directly mapping onto the corresponding PM2 functions.

3.1 Threads and Communication

Threads Subsystem. The threads component of Hyperion is a very thin layer that interfaces to Marcel (PM2's thread library). Marcel is an efficient, user-level, POSIX-like thread package featuring thread migration. Most of the functions in Marcel's API provide the same syntax and semantics as the corresponding POSIX Threads functions. However, it is important to note that the Hyperion thread component uses the PM2 thread component through the PM2 API and does not access the thread component directly, as would be typical when using a classical *Pthreads*-compliant package. PM2 implements a careful integration of multithreading and communication that actually required several modifications of the thread management functions (e.g., thread creation). Thus, it would be inefficient (and even dangerous) to bypass the PM2 API by using the underlying thread package directly.

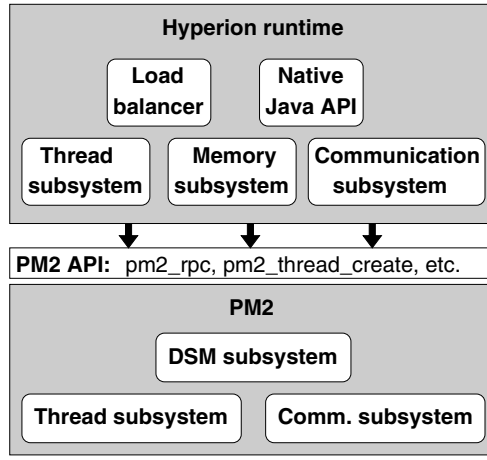


Fig. 2. Overview of the Hyperion software architecture

Communication Subsystem. The communication component of Hyperion is implemented using PM2 remote procedure calls (RPCs), which allow PM2 threads to invoke the remote execution of user-defined services (i.e., functions). On the remote node, PM2 RPC invocations can either be handled by a pre-existing thread or they can involve the creation of a new thread. This latter functionality allows us to easily implement Hyperion's communication subsystem. PM2 utilizes a generic communication package [3] that provides an efficient interface to a wide-variety of high-performance communication libraries, including low-level ones. The following network protocols are currently supported: BIP (Myrinet), SISCI (SCI), VIA, MPI, PVM and TCP.

3.2 Memory Management

The memory management primitives described in Table 1 are implemented on top of PM2's distributed-shared-memory layer, DSM-PM2 [1]. DSM-PM2 has been designed to be generic enough to support multiple consistency models. Sequential consistency and Java consistency are currently available. Moreover, for a given consistency model, alternative protocols (based on page migration and/or on thread migration) are provided. Also, new consistency models can be easily implemented using the existing generic DSM-PM2 library routines.

DSM-PM2 is structured in layers. At the high level, a *DSM protocol policy* layer is responsible for implementing consistency models out of a subset of the available library routines and for associating each application data with its own consistency model. The library routines (used to bring a copy of a page to a thread, to migrate a thread to a page, to invalidate all copies of a page, etc.) are grouped in the lower-level *DSM protocol library* layer. Finally, these library routines are built on top of two base components: the *DSM page manager* and the *DSM communication module*. The *DSM page manager* is essentially dedicated

to the low-level management of memory pages. It implements a distributed table containing page ownership information and maintains the appropriate access rights on each node. The *DSM communication module* is responsible for providing elementary communication mechanisms, such as delivering requests for page copies, sending pages, and invalidating pages.

The DSM-PM2 user has three alternatives that may be utilized according to the user's specific needs: (1) use a built-in protocol, (2) build a new protocol out of a subset of library routines, or (3) write new protocols using the API of the *DSM page manager* and *DSM communication module* (for more elaborate features not implemented by the library routines). The Hyperion DSM primitives (`loadIntoCache`, `updateMainMemory`, `invalidateCache`, `get` and `put`) have been implemented using this latter approach.

Object replication: main memory and caches. To implement the concept of *main memory* specified by the Java model, the run-time system associates a *home node* to each object. The home node is in charge of managing the reference copy. Initially, the objects are stored on their home nodes. They can be replicated if accessed on other nodes. Note that at most one copy of an object may exist on a node and this copy is shared by all the threads running on that node. Thus, we avoid wasting memory by associating caches to nodes rather than to threads.

Access detection and modification recording. Hyperion uses specific access primitives to shared data (`get` and `put`), which allows us to use explicit checks to detect if an object is present (i.e., has a copy) on the local node. If the object is present, it is directly accessed, else the page(s) containing the object is locally cached. Thanks to the access primitives, the modifications can be recorded at the moment when they are carried out. For this purpose, a bitmap is created on a node when a copy of the page is received. The `put` primitive uses it to record all writes to the object, using *object-field granularity.* All local modifications are sent to the home node of the page by the `updateMainMemory` primitive.

Implementing objects on top of pages. Java objects are implemented on top of DSM-PM2 pages. If an object spans several pages, all the pages are cached locally when the object is loaded. Consequently, loading an object into the local cache may generate prefetching, since all objects on the corresponding page(s) are actually brought to the current node. Similarly, when updating the master copy of an object, other objects located on the same page will get their master copy updated. This implementation has been carefully designed to be fully compliant with Java consistency [9].

Object ownership. Our implementation allocates all Java objects within the section of memory controlled by DSM-PM2. We align Java objects on 2^k-byte boundaries. An object reference is basically the address of the object, but now we can use the bottom k bits to store the node number of the owner of the object. (k can be adjusted to accommodate larger number of nodes, at the expense of increasing internal fragmentation.) This allows us to do an efficient ownership test in the `loadIntoCache` primitive with a bitwise

AND, a subtract, and a test against zero. If the ownership test fails, then the DSM-PM2 page table is consulted to see if the page containing the object is locally cached. In addition, each object is given a standard header and one of the bits of the header is used to indicate if the object is a cached copy or not. This bit is used by the `put` primitive to quickly determine whether the page is locally owned or not. If not, the modification needs to be recorded in the bitmap associated with the DSM-PM2 page holding the cached copy of the object.

4 Performance Evaluation: Minimal-Cost Map-Coloring

4.1 Experimental Conditions and Benchmark Programs

We have implemented branch-and-bound solutions to the minimal-cost map-coloring problem, using serial C, serial Java, and multithreaded Java. These programs have first been run on a eight-node cluster of 200 MHz Pentium Pro processors, running Linux 2.2, interconnected by a Myrinet network and using MPI implemented on top of the BIP protocol [15]. We have also executed the programs *without any modification* on a four-node cluster of 450 MHz Pentium II processors running Linux 2.2, interconnected by a SCI network using the SISCI protocol.

The serial C program is compiled using the GNU C compiler, version 2.7.2.3 with `-O6` optimization, and runs "natively" as a normal Linux executable. The Java programs are translated to C by Hyperion's `java2c` compiler, the generated C code is also compiled by GNU C with `-O6` optimization, and the resulting object files are linked with the Hyperion/PM2 run-time system.

The two serial programs use identical algorithms, based upon storing the search states in a priority queue. The queue first gives priority to states in the bottom half of the search tree and then secondly sorts by bound value. (Giving priority to the states in the bottom half of the search tree drives the search to find solutions more quickly, which in turn allows the search space to be more efficiently pruned.)

The parallel program does an initial, single-threaded, breadth-first expansion of the search tree to generate sixty-four search states. Sixty-four threads are then started and each one is given a single state to expand. Each thread keeps its own priority queue, using the same search strategy as employed by the serial programs. The best current solution is stored in a single location, protected by a Java monitor. All threads poll this location at regular intervals in order to effectively prune their search space.

Maintaining a constant number of threads across executions on different size clusters helps to keep fairly constant the aggregate amount of work performed across the benchmarking runs. However, the pattern of interaction of the threads (via the detection of solutions) does vary and thus the work performed also varies slightly across different runs.

All programs use a pre-allocated pool of search-state data objects. This avoids making a large number of calls to either the C storage allocation primitives

(`malloc`/`free`) or utilizing the Java garbage collector. (Our distributed Java garbage collector is still under development.) If the pool is exhausted, the search mechanism switches to a depth-first strategy until the pool is replenished.

For benchmarking, we have solved the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Assigning sixty-four threads to this problem in the manner described above, and using Hyperion's round-robin assignment of threads to nodes, is reasonably effective at evenly spreading the number of state expansions performed around a cluster, if the number of nodes divides evenly into the number of threads. (In the future we plan to investigate dynamic and transparent load balancing approaches that utilized the thread migration features of PM2.)

4.2 Overhead of Hyperion/PM2 vs. Hand-Written C Code

First, we compare the performance of the serial programs on a *single* 450 MHz Pentium II processor running Linux 2.2. Both the hand-written C program and the C code generated by `java2c` were compiled to native Pentium II instructions using `gcc2.7.2.3` with option `-O6`. Execution times are given in seconds.

Hand-written C	63
Java via Hyperion/PM2	324
Java via Hyperion/PM2, in-line DSM checks disabled	168
Java via Hyperion/PM2, array bound checks also disabled	98

We consider this a "hard" comparison because we are comparing against hand-written, optimized C code and because the amount of straight-line computation is minimal. This application features a large amount of object manipulation (inserting to and retrieving from the priority queue; allocating new states from the pool and returning "dead-end" states to the pool) with a relatively small amount of computation involved in expanding and evaluating a single state. In fact, the top two lines in the table demonstrate that the *original* Hyperion/PM2 execution is roughly five times slower than the execution of the hand-written C program.

The bottom two lines in the table help explain the current overheads in the Hyperion/PM2 implementation of the Java code and represent cumulative improvements to the performance of the program. In the third line of the table, the Java code is executed on a single node with the *in-line checks* disabled: in-line checks are used by Hyperion to test for the presence or the absence of a given Java object at the local node in the distributed implementation; as there is only one node at work in the case at stake, they are always satisfied. In the fourth line of the table, the array bound checks are additionally disabled. This last version can be considered as the closest to the hand-written C code. It is only 55% slower. A comparison with hand-written C++ code would probably be more fair to Hyperion, and would probably result in an even lower gap.

We can draw two conclusions from these figures. First, the in-line checks used to implement the Hyperion DSM primitives (e.g., `loadIntoCache`, `get` and `put`)

are very expensive for this application. By disabling these checks in the C code generated by `java2c`, we save nearly 50% of the execution time. (This emphasizes the map-coloring application's heavy use of object manipulation and light use of integer or floating-point calculations.) For this application it may be better to utilize a DSM-implementation technique that relies on page-fault detection rather than in-line tests for locality. This can be easily done within the context of DSM-PM2's generic support and we are currently evaluating this alternative, i.e., *in-line* vs. *page-fault* checks, with an expanded set of applications.

Second, the cost of the array-bounds check in the Java array-index operation, at least in the Hyperion implementation, is also quite significant. We implement the bounds check by an explicit test in the generated C code. In the map-coloring application, arrays are used to implement the priority queues and in the representation of search states. In both cases the actual index calculations are straightforward and would be amenable to optimization by a compiler that supported the guaranteed safe removal of such checks. Such an optimization could be implemented in `java2c`. Alternatively this optimization might be done by analysis of the bytecode prior to the execution of `java2c`. Or, the optimization could be performed on the generated C code. We plan to further investigate these alternatives in the future.

4.3 Performance of the Multithreaded Version

Next, we provide the performance of the multithreaded version of the Java program on the two clusters described in Section 4.1. Parallelizability results are presented in Figure 3: the multi-node execution times are compared to the single-node execution time of the same multithreaded Java program run with 64 threads.

On the 200 MHz Pentium Pro cluster using MPI-BIP/Myrinet, the execution time decreases from 638 s for a single-node execution to 178 s for an 4-node execution (90% efficiency), and further to 89 s for an 8-node execution (still 90% efficiency). On the 450 MHz Pentium II cluster using SISCI/SCI, the efficiency is slightly lower (78% on 4 nodes), but the execution time is significantly better: the program runs in 273 s on 1 node, and 89 s on 4 nodes. Observe that the multithreaded program on one 450 MHz Pentium II node follows a more efficient search path for the particular problem being solved than its serial, single-threaded version reported in Section 4.1.

The efficiency decreases slightly as the number of nodes increases on a cluster. This is due to an increasing number of communications that are performed to the single node holding the current best answer. With a smaller number of nodes, there are more threads per node and a greater chance that, on a given node, requests by multiple threads to fetch the page holding the best answer can be satisfied by a single message exchange on the network. (That is, roughly concurrent page requests at one node may be satisfied by one message exchange.)

We believe these results indicate the strong promise of our approach. However, further study is warranted. We plan to investigate the performance under

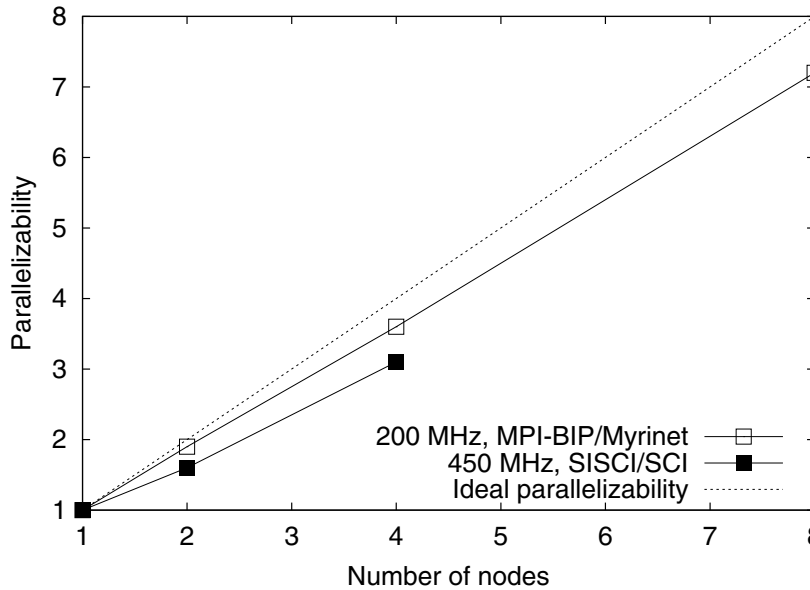


Fig. 3. Parallelizability results for the multithreaded version of our Java program solving the problem of coloring the twenty-nine eastern-most states in the USA using four colors with different costs. Tests have been done on two cluster platforms: 200 MHz Pentium Pro using MPI-BIP/Myrinet and 450 MHz Pentium II using SISCI/SCI. The program is run in all cases with 64 threads.

Hyperion/PM2 of additional Java multithreaded programs, including applications converted from the SPLASH-2 benchmark suite.

5 Related Work

The use of Java for distributed parallel programming has been the object of a large number of research efforts during the past several years. Most of the recently published results highlight the importance of *transparency* with respect to the possibly distributed underlying architecture: multithreaded Java applications written using the shared-memory paradigm should run unmodified in distributed environments. Though this goal is put forward by almost all distributed Java projects, many of them fail to fully achieve it.

The JavaParty [13] platform provides a shared address space and hides the inter-node communication and network exceptions internally. Object and thread location is transparent and no explicit communication protocol needs to be designed nor implemented by the user. JavaParty extends Java with a preprocessor and a run-time system handling distributed parallel programming. The source code is transformed into regular Java code plus RMI hooks and the

latter are fed into Sun's RMI compiler. Multithreaded Java programs are turned into distributed JavaParty programs by identifying the classes and objects that need to be spread across the distributed environment. Unfortunately, this operation is not transparent for the programmer, who has to explicitly use the keyword `remote` as a class modifier. A very similar approach is taken by the Do! project [10], which obtains distribution by changing the framework classes used by the program and by transforming classes to transparently use the Java RMI, while keeping an unchanged API. Again, potentially remote objects are explicitly indicated using the `remote` annotation.

Another approach consists in implementing Java interpreters on top of a distributed shared memory [6, 17] system. Java/DSM [17] is such an example, relying on the Treadmarks distributed-shared-memory system. Nevertheless, using an off-the-shelf DSM may not lead to the best performance, for a number of reasons. First, to our knowledge, no available DSM provides specific support for Java consistency. Second, using a general-purpose release-consistent DSM sets up a limit to the potential specific optimizations that could be implemented to guarantee Java consistency. Locality and caching are handled by the DSM support, which is not flexible enough to allow the higher-level layer of the system to configure its behavior.

cJVM [2] is another interpreter-based JVM providing a single image of a traditional JVM while running on a cluster. Each cluster node has a cJVM process that implements the Java interpreter loop while executing part of the application's Java threads and containing part of the application objects. In contrast to Hyperion's object caching approach, cJVM executes methods on the node holding the master copy of the object, but includes optimizations for data caching and replication in some cases.

Our interest in computationally intensive programs that can exploit parallel hardware justifies three main original design decisions for Hyperion. First, we rely on a Java-to-C compiler to transform bytecode to native code and we expect the compilation cost will be recovered many times over in the course of executing such programs. We believe this approach will lead to much better execution times compared to the interpreter-based approaches mentioned above. Second, Hyperion uses the generic, multi-protocol DSM-PM2 run-time system, which is configured to specifically support Java consistency. Finally, we are able to take advantage of fast cluster networks, such as SCI and Myrinet, thanks to our portable and efficient communication library provided by the PM2 run-time system.

6 Conclusion

We propose utilizing a cluster to execute a single Java Virtual Machine. This allows us to run Java threads completely transparently in a distributed environment. Java threads are mapped to native threads available on the nodes and run with true concurrency. An original feature of our system is its use of a Java-to-C compiler (and hence of machine code). Hyperion's implementation supports a

globally shared address space via the DSM-PM2 run-time system that we configured to guarantee Java consistency. The generic support provided by DSM-PM2 allowed us to implement Java-specific optimizations that are not available in standard DSM systems, such as Treadmarks. Thanks to the portability of the PM2 run-time support, the full system we present is available on top of several UNIX systems and can use a large variety of network protocols. To evaluate our approach, we compare serial C, serial Java, and multithreaded Java implementations of a branch-and-bound solution to the minimal-cost map-coloring problem. We report good parallelizability on two platforms using two different network protocols: SISCI/SCI and MPI-BIP/Myrinet.

References

[1] Gabriel Antoniu, Luc Bougé, and Raymond Namyst. Generic distributed shared memory: the DSM-PM2 approach. Research Report RR2000-19, LIP, ENS Lyon, Lyon, France, May 2000.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.

[3] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–182, San Juan, Puerto Rico, April 1999. Held in conjunction with IPPS/SPDP 1999., Springer-Verlag.

[4] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 91–100, Palo Alto, California, February 1998.

[5] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10:1125–1242, 1998.

[6] X. Chen and V. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1998.

[7] A. Ferrari. JPVM: Network parallel computing in Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 245–249, Palo Alto, California, 1998.

[8] V. Getov, S. Flynn-Hummell, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 45–54, Palo Alto, California, February 1998.

[9] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addision-Wesley, Reading, Massachusetts, 1996.

[10] P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *High-Performance Computing and Networking (HPCN '98)*, volume 1401 of *Lect. Notes in Comp. Science*, pages 628–637. Springer-Verlag, 1998.

[11] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Third Usenix Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[12] R. Namyst and J.F. Mehaut. PM^2 : Parallel Multithreaded Machine: A computing environment for distributed architectures. In *ParCo'95 (Parallel Computing)*, pages 279–285. Elsevier Science Publishers, September 1995.

[13] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.

[14] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Third Usenix Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997.

[15] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on Myrinet. In *Proceedings of First Workshop on Personal Computer Based Networks Of Workstations*, volume 1388 of *Lect. Notes in Comp. Science*. Springer-Verlag., April 1998.

[16] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *International Symposium on Computer Architectures*, pages 256–266, Gold Coast, Australia, May 1992.

[17] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of the Workshop on Java for High-Performance Scientific and Engineering Computing*, Las Vegas, Nevada, June 1997.

A More Expressive Monitor for Concurrent Java Programming[1]

Hsin-Ta Chiao, Chi-Houng Wu, and Shyan-Ming Yuan

Department of Computer and Information Science
National Chiao Tung University
1001 Ta Hsueh Rd., Hsinchu 300, Taiwan

{gis84532, gis86501, smyuan}@cis.nctu.edu.tw

Abstract. The thread synchronization mechanism employed by Java is derived from the Hoare's monitor concept. In order to minimize its implementation complexity, the monitor provided by Java is quite primitive. This design decision prefers simple concurrent objects and single-thread programs. However, we think the Java monitor is over simplified for developing more elaborated concurrent objects. Besides, several features of the Java monitor will bring extra overhead when thread contention gets higher. Currently, we have identified five drawbacks of the Java monitor. In this paper, we will first analyze these drawbacks in depth, and then propose a new monitor-based synchronization mechanism called EMonior. It has better expressive power, and introduces fewer overheads than the Java monitor when contention occurs. EMonior uses a preprocessor to translate the Java programs containing EMonitor syntax to regular Java programs that invoke EMonitor class libraries. It is very suitable to replace the Java monitor with the EMonitor when developing elaborate concurrent objects or high-contention concurrent systems.

1 The Introduction to Java Monitor

The thread synchronization mechanism offered by Java is a monitor [5], which is a simplification of the original Hoare's monitor concept [8]. For implementing monitors, each Java object contains a monitor lock and a condition queue. The keyword *synchronize* can be inserted into a method's definition for specifying it as a synchronized method. In each Java object, there is always only one synchronized method can be running at any moment. In addition to synchronized methods, Java also offers synchronized blocks for reducing the size of critical sections. Besides, for condition synchronization, Java provides the following three methods in each object: ***wait***( ), ***notify***( ), and ***notifyAll***( ). They can be invoked only inside a synchronized method or inside a synchronized block.

The design philosophy of Java monitor is to keep it as simple as possible. Consequently, its simplicity also leads to efficient implementation. All concurrent objects that can be implemented by the Java monitor easily will benefit from this design principle. Besides the multi-thread, concurrent programs, this design principle

[1] This work was supported both by the National Science Council grant NSC88-2213-E-009-087 and the industry reaearch program 89-EC-2-A-17-0285-006 of the ROC Economic Bureau.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1053-1060, 2000.
© Springer-Verlag Berlin Heidelberg 2000

will also reduce the run-time overhead of single-thread programs. This is because Java offers only one suite of class libraries to both single-thread and multi-thread programs. Hence, all classes in the class libraries have to employ the Java monitor if necessary to guarantee the thread-safe property. Once the Java monitor's overhead is shrunk, both the single-thread and multi-thread programs will have a more efficient class libraries. In addition, by employing the thin-lock approach that has first been implemented in JDK 1.1.2 for AIX [3], the Java monitor lock can be further optimized for the situation where no contention occur. The similar techniques are currently employed by many Java virtual machines. Hence, if the required concurrent objects are simple and the contention level of the execution environment is low, the current design of Java monitor seems very successful.

However, for designing elaborate concurrent Java objects, we think that the Java monitor is over simplified and lacks of expressive power. Besides, several features of the Java monitor will bring extra overhead when the contention between the cooperative threads gets higher. Consequently, using Java monitor in the above scenarios may both complicate program design and produce less efficient programs. In this paper, we will first analyze the five known drawbacks of Java monitor in section 2. Then, in section 3, we will propose a new monitor-based thread synchronization mechanism called EMonitor for replacing the Java monitor. The performance comparison between the Java monitor and EMonitor are shown in section 4. Last, section 5 concludes this paper.

2 The Drawbacks of Java Monitor

2.1 The Problems Introduced by Single Condition Queue

As introduced in section 1, there is only one condition queue in a Java object. If all pending threads in a condition queue wait for the same event, no further problem will occur. However, if this preferred property is spoiled, the event triggering a ***notify***( ) method may be different from the event required by the thread waked by the ***notify***( ). For solving this problem, the awaked thread should check whether the required event has happened. If not, it will first use ***notify***( ) to resume another pending thread in the condition queue, and then invoke ***wait***( ) to block itself again. The above steps are repeated until a correct pending thread is picked. Another solution is to replace ***notify***( ) with ***notifyAll***( ) to resume all the pending threads at once. Each awaked thread has to determine whether the required event is happened. However, these two approach both wake up many irrelevant pending threads, and extra thread context switches are introduced. This overhead is even worse if a Java virtual machine directly maps a Java thread to an operating system's kernel thread. [6].

Several programming techniques [12] or design patterns [10] have been proposed for solving the Java monitor's single-condition-queue problem. All these solution use extra supporting objects to simulate the presence of multiple condition queues inside a Java object. However, we find that designing a concurrent Java object by cooperating with these supporting objects is complex, error-prone and deadlock-prone. Hence, we

think it is still necessary to provide multiple condition queues inside a monitor at the programming language level directly.

2.2 No Additional Support for Scheduling

Scheduling means to choose a preferred request among a group of incoming requests. This also implies the scheduler must possess some global information about these requests for scheduling decision. For example, many existent monitors [1][11][13][14] offer a prioritized condition queue to simplify the task of writing static scheduling programs. However, Java monitor barely provides a random condition queue, and does not support any other facilities that can help a program to maintain the global information for scheduling. Consequently, a Java program has to keep the required scheduling information in other more expensive ways.

2.3 The Troubles Caused by No-Priority Monitor

The behavior of different monitors can be classified by a generalized queue model [2]. In this model, a monitor lock contains three queues: an entry queue (holding the incoming threads), a waiting queue (holding the signalled or the notified threads), and a signaller queue (holding the signaller or the notifying thread). For preventing starvation, the entry queue's priority ***Ep*** should always be the lowest. If a monitor's ***Ep*** is equal to either the ***Sp*** (the signaller queue's priority) or the ***Wp*** (the waiting queue's priority), we refer to it as a *no-priority monitor*. Otherwise, it is a *priority monitor*. Because the Java monitor use non-blocking signal semantic, its monitor lock contains no signaller queue. Furthermore, since the property of the lock waiting queue is left unspecified in the Java specification [5], it is reasonable to assume Java monitor's ***Ep*** and ***Wp*** are equal. Hence, we can classify the Java monitor as a no-priority one.

However, no-priority monitors have two problems. The first problem is ***wait***( )'s post-condition may be different from the corresponding ***notify***( )'s pre-condition [1][2][12]. In fact, this condition-breakup problem is caused by other imcoming threads that preempt the notified thread, and enter the monitor to destroy the ***notify***( )'s pre-condition. Once the ***wait***( ) returns, if its post-condition is identical to the *notify*( )'s pre-condition, the statements behind it are allowed to proceed. Otherwise, the notified thread has to invoke ***wait***( ) again to block itself. This will cause extra context switches and sink the program performance.

In addition to the condition-breakup problem, no-priority monitors may also complicate the task of writing scheduling programs [1][2][7]. In a scheduler implemented by a monitor, a calling thread often judges whether itself can be scheduled immediately at the beginning of a monitor method. If it cannot, it will block itself on the condition queue. Then, when a pending thread in the condition queue can be scheduled to run, it will be notified by another thread, and stay in waiting queue for entering the monitor again. However, under some boundary conditions (for instance, the condition queue becomes empty after it is notified), due to the property of no-priority monitors, the notified thread may be preempted by the threads in the entry queue. This disturbance will cause the execution order of threads

to violate the desired scheduling rule. Consequently, if no-priority monitors are adopted to write scheduling programs, programmers have to consider the above interference.

2.4 Insufficient Signal Semantics

In a Java monitor, both ***notify***( ) and ***notifyAll***( ) do not release the monitor lock, and they merely resume either one or all pending threads in the condition queue, respectively. Hence, the semantics of both ***notify***( ) and ***notifyAll***( ) belong to the non-blocking signal [2][12]. However, besides the non-blocking signal, there are other practical signal semantics. For instance, if blocking signal [2] is employed, ***wait***( )'s post-condition will always match the corresponding ***notify***( )'s pre-condition. Therefore, the blocking-signal monitor is easier to use. However, the performance of programs written in the non-blocking signal is better than the ones written in the blocking signal. Hence, good performance is the chief advantage of the non-blocking signal. In addition, the immediate-return signal semantic is also worthy to be considered. We thought the *signal and exit* action of this semantic is useful. It can reduce the overhead introduced by a thread that calls ***notify***( ) and then instantly leaves the monitor. Another kind of action offered by the immediate-return signal semantic is the *signal and wait* action. Since implementing this action will cause different kinds of condition queues (if more than one kind of condition queue is offered) depend on each other, we suggest to abandon it.

2.5 Deadlock of Inter-monitor Nested Calls

Java uses nested mutually exclusive locks to implement synchronized methods. Consequently, deadlock will not occur during an intra-monitor nested call. However, deadlock may arise in inter-monitor nested calls. This can be further divided into two categories. The first kind of deadlock is the mutually-dependent deadlock, which is well known and has been figured out in [4][12][15]. The second kind of deadlock is the condition-wait deadlock [9]. For preventing the deadlock arise in inter-monitor nested calls, the open call semantic [1][9] can be employed. Before a thread performs an open inter-monitor nested call, it will release the caller monitor lock. After the inter-monitor nested call returns, this thread will acquire the monitor lock again for reentering the caller monitor. Since each thread always holds at most one monitor lock, the mutually-dependent deadlock and the condition wait deadlock are eliminated. However, the open calls enforce extra restrictions on the programming style. Before invoking an open call, the calling thread has to transfer the caller monitor's state to a consistent state, in which all the monitor invariants are true.

3 Our Solution

In this section, we will propose a new monitor-based synchronization mechanism – the EMonitor. It is designed to replace the Java monitor for constructing elaborate concurrent Java objects, or for building the concurrent systems with higher contention level.

3.1 The Characteristics of the EMonitor

The EMonitor is a priority monitor. It supports multiple signal semantics within a monitor. These semantics are blocking signal, non-blocking signal (include non-blocking signal all), and the signal and exit action of the immediate-return signal semantic. In addition, EMonitor provides two different kinds of condition queues – the prioritized condition queues and the FIFO condition queues. The FIFO condition queues are very generalized and have enough expressive power to deal with more complex dynamic scheduling problems. Of course, there can be more than one condition queue within an EMonitor object. Since preventing deadlock and keeping the programming style simple are conflicting requirements, we determine to provide both the open calls and the closed calls (the original semantic of Java's inter-monitor nested call.)

3.2 The Syntax and Implementation of EMonitor

Our EMonitor can be used in three ways, and each of them has a different granularity of mutual exclusion. The first way is through the *EMonitored classes*. We add a new class modifier, *EMonitored*, to Java. If it is inserted into the declaration of a Java class, this class will become an EMonitored class. In an instance of an EMonitored class, only one method invocation can be running at the same time. The other two ways of using EMonitor are through the *EMonitored methods* and the *EMonitored blocks*. The syntax and functionality of these two constructs are very similar to the synchronized methods and synchronized blocks of Java (except the keyword becomes *EMonitored*.)

We treat the EMonitor as a language extension to Java. A preprocessor is responsible for translating a Java program using the EMonitor to a pure Java program that invokes the EMonitor class libraries. Both the preprocessor and the EMonitor class libraries are implemented only by Java itself. Consequently, the Java virtual machine requires no modification. The EMonitor class libraries contains five primary classes, and they are the **EMonitor** class, the **EMonitoredThread** class, the **Condition** class, the **PrioritizedCondition** class, and the **FIFOCondition** class. The **EMonitor** class implements a nested lock – the EMonitor lock. Unlike the Java monitor lock, it contains an extra queue - the return queue. It holds the threads return from the issued open calls. The **EMonitoredThread** class itself is a subclass of the Java's **Thread** class. It contains an extra stack to store the EMonitor locks' nested count for each unreturned open call. Hence, each thread using the EMonitor has to be an instance of the **EMonitoredThread** class or its subclass.

As mentioned in the previous subsection, our EMonitor provides two kinds of condition queues. The **Condition** class is an abstract class that defines the common properties for both the prioritized and FIFO condition queues. The **PrioritizedCondition** class is for the prioritized condition queues The **FIFOCondition** class is for the FIFO condition queues. In a FIFO condition queue, each pending thread can be associated with a customized object that is specified when the thread invokes the ***Wait***( ) methods. Therefore, the scheduling information about a pending thread can be stored in the object attached with it. Besides, a specialized

Wait*(Object** *attribute*, int *n*) method is also offered. It can insert the calling thread into a designated location (indexed by *n*) of a FIFO condition queue. For retrieving the scheduling information, the ***Dump( ) and the ***Peek***( ) method are provided. The ***Swap***( ) function is offered to swap the position of two pending threads. Besides, for all signal semantics except the non-blocking signal all, the corresponding indexed signal methods are provided in this class. Consequently, we can signal a specific pending thread through one of these methods by assigning its index in a condition queue.

4 Experimental Result

In this section, we will compare the performance of our EMonitor with the Java monitor. We first construct two multiple-reader/single-writer locks. One is implemented by the EMonitor, and another is implemented by the Java monitor. Then, we measure the lock overhead introduced by each kind of implementation.

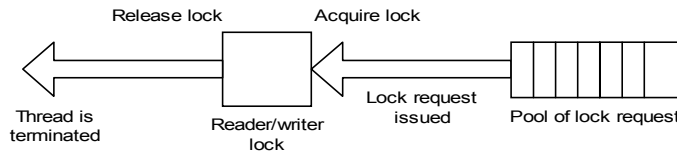
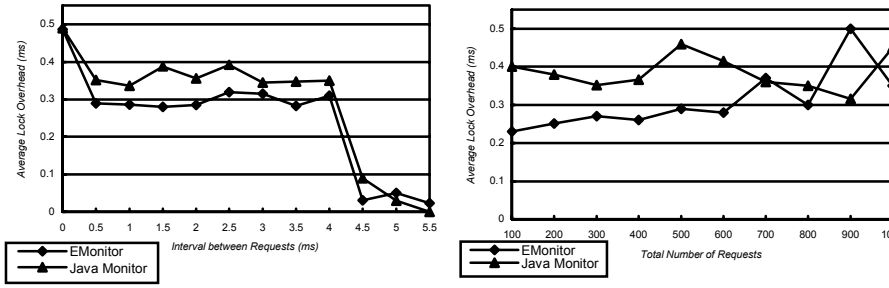


Fig. 1. The experiment for measuring the average lock overhead

Please refer to Figure 1. All the necessary threads are created in advance, and then put into the lock request pool. When the experiment starts, the lock requests (threads) will be issued from the lock request pool in a one-by-one fashion. Each issued thread intends to acquire an assigned lock from the reader/writer lock, and we always fix the ratio of read locks to write locks as 6:4. Once the desired lock is granted, the issued thread will perform an operation, which takes about 3.9 ms. After this operation is finished, the lock held by this thread will be released. In an experiment, we measure the elapsed time of each thread since it starts to acquire a lock, until it releases the lock. Then, we calculate the average elapsed time per each thread.

In this section, we conduct two different kinds of experiments to reveal the properties of both the EMonitor and the Java monitor. In the first experiment, we fix the number of issued lock requests as 500. Then, we observe the change in the lock overheads by altering the time interval for issuing a new lock request. The curve of the measured lock overheads is shown in Figure 2(a). In the second experiment, we fix the interval between lock requests as 1 ms, and vary the total number of lock requests. Furthermore, this short interval will cause all issued threads (except the first one) to wait for the desired locks. Figure 2(b) shows the measured lock overheads of this experiment. In this section, all the experimental results are gathered in an unloaded, 300MHz Sun UltraSPARC-II machine. The operation system is Sun Solaris 2.6, and the version of JDK is 1.1.3.



(a) **(b)**

Fig. 2 (a) shows the experimental results for fixing the total number of lock requests. (b) shows the experimental results for fixing the time interval between lock requests.

In Figure 2(a), we can observe that the lock overheads increase drastically once the time interval is reduced to 4 ms. This overhead boost also implies the synchronization contention occurs. In this figure, when contention is present, the EMonitor's average lock overhead is about 15% less than the Java monitor's lock overhead. In contrast, if no contention happens, the Java monitor's lock overhead will become smaller than the EMonitor's. This result is reasonable since the EMonitor itself is implemented by the Java monitor. In Figure 2(b), the short time interval causes contention always to occur. Consequently, the EMonitor's average lock overhead is about 19.4% less than the Java monitor's.

From both Figure 2(a) and Figure 2(b), we can find that once contention is present, in most circumstances, the EMonitor's lock overhead is smaller than the Java monitor's. The reason is stated as follows: When a thread releases the lock it holds, the Java monitor's reader/writer lock will invoke ***notifyAll***( ) to wake up all pending reader and pending writer threads. Then, each awaked thread will enter the monitor, and determine whether it can get the desired lock or not. Consequently, extra thread context switches are introduced, and the Java monitor's average lock overhead is increased. On the other hand, the EMonitor can avoid this problem by providing more precise event notification.

5 Conclusion

In this paper, we first analyze the five known drawbacks of the Java's monitor in detail. Then, we proposed our solution – the EMonitor. It is still monitor-based, and can solve all Java monitor's problems we have identified. The EMonior offers better expressive power. Besides, when contention is present, the concurrent programs implemented by the EMonitor are more efficient than the ones implemented by the Java monitor. The experimental result in section 4 confirms our idea. Hence, we suggest replacing the Java monitor with the EMonitor when developing elaborate concurrent objects or high-contention concurrent systems.

References

1. G. Andrews, *Concurrent Programming - Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc., 1991.
2. P. Buhr, M. Fortier, and M. Coffin, "Monitor Classification," *ACM Computing Surveys*, vol. 27, no. 1, pp. 63-107, March 1995.
3. D. Bacon, R. Konura, and C. Murthy, et al, "Thin Locks: Featherweight Synchronization for Java," *Proc. ACM SIGPLAN '98 Conf. on Programming Language Design and Implementation*, pp. 258-268, 1998.
4. B. Brosgol, "A Comparison of the Concurrency Features of Ada 95 and Java," *Proc. of ACM SIGAda Annual Int'l Conf. on Ada Technology*, pp. 175-192, 1998.
5. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.
6. Y. Gu, B. Lee, and W. Cai, "Evaluation of Java Thread Performance on Two Differenct Multithreaded Kernels," *Operating Systems Review*, vol. 33, no. 1, pp. 34-46, Jan. 1999.
7. N. Gehani, "Capsules: A Shared Memory Access Mechanism for Concurrent C/C++," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 795-811, July 1993.
8. C. Hoare, "Monitor: An Operating System Constructing Concept," *CACM*, vol. 17, no. 10, pp. 549-557, 1974.
9. L. Kotulski, "About the Semantic Nested Monitor Calls," *SIGPLAN Notices*, vol. 22, no. 4, pp. 80-82, 1987.
10. D. Lea, *Concurrent Programming in Java – Design Principles and Patterns*, Addison-Wesley, 1997.
11. 11. R. Olsson and C. McNamee, "Experience Using the C Preprocessor to Implement CCR, Monitor, and CSP Preprocessors for SR," *Software - Practice and Experience*, vol. 26, no. 2, pp.125-134. Feb. 1996.
12. S. Oaks and H. Wong, *Java Threads*, O'Reilly & Associates, Inc., 1997.
13. S. Stubbs, D. Carver, and A. Hoppe, "IPCC++: A Concurrent C++ Based on a Shared-Memory Model," *Journal of Object-Oriented Programming*, vol. 8, no.2, pp. 45-50, 66, May 1995.
14. S. Yuan and Y. Hsu, "Design and Implementation of a Distributed Monitor Facility," *Computer Systems Science and Engineering*, vol. 12, no. 1, pp. 43-51, Jan. 1997.
15. C. Varela and G. Agha, "What after Java? From Objects to Actors," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp.573-577, 1998.

An Object-Oriented Software Framework for Large-Scale Networked Virtual Environments

Frédéric Dang Tran and Anne Gérodolle

France Telecom R&D DTL/ASR, 38-40 rue du Général Leclerc,
92794 Issy Moulineaux Cedex 9, FRANCE
{frederic.dangtran, anne.gerodolle}@francetelecom.fr

Abstract. Continuum is an object-oriented software framework that aims to offer an open and extensible foundation for building large-scale real-time networked virtual environment applications. This platform relies on a partial replication model in which the whole simulation space is spread on a federation of processes based on application-specific criteria. A configurable event communication framework allows arbitrary consistency and synchronization protocols to be implemented in close cooperation with the application semantics. Continuum has been experimented with success for multi-player game prototypes involving both human-driven and autonomous simulated entities.

1 Introduction

With the tremendous success of the Internet, multi-participant real-time distributed simulations based on a shared "virtual world" paradigm are bound to develop at a fast pace: multi-user online games, virtual shopping malls, concurrent design applications etc. These applications attempt to provide to end-users the illusion of being immersed in a conceptually unique shared virtual environment where they can (usually by way of their so-called avatar) interact in real-time with other users or computer-controlled objects.

Large-scale shared virtual world applications raise many challenges. Foremost among them is the necessity to provide a globally consistent virtual environment to end-users. Problems typical of distributed systems such as synchronization, consistency and availability are made all the more difficult by the real-time and scalability requirements of large-scale human-in-the-loop simulations. A user may initiate an event, such as modifying an object in the 3D scene, and expects the effect of that action to be visible within a bounded time.

The Continuum platform presented in this paper aims to offer an open and extensible object-oriented architecture which meets the following requirements:

- support for shared spaces involving several hundreds or thousands of simultaneous participants spread over a vast geographic area;
- support for dynamic and persistent virtual environments which are sustained and "alive" even when all human participants have left;

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1061–1070, 2000.
© Springer-Verlag Berlin Heidelberg 2000

– support for highly interactive shared spaces populated by both user-controlled and autonomous objects interacting with one another in real-time;

Instead of proposing a one-size-fits-all platform, the Continuum platform is based on an open software framework from which several "profiles" can be derived (e.g. for collaborative applications, for real-time online games etc.), each addressing a specific application domain. Our design approach is to some extent similar to Application Level Framing approach [3] proposed for building communication protocols. Our platform enables the application programmer to integrate closely the semantics of the application with the mechanisms of the platform.

The rest of this paper is outlined as follows. Section 2 describes the object and perception model which lies as the foundation of our architecture. The next section explains the replication model and its implication on the scalability of the platform. Section 4 describes the event model of the platform. The platform architecture is outlined in section 5. Related work is summarized in section 6. Finally we conclude and point to future directions.

The Continuum platform is strongly Java technology centered and assumes that the Java programming language is used both at the application and system level. All code fragments provided in this paper are in Java.

2 Object and Perception Model

A shared information space consists of a collection of objects. Each of these object encapsulates a state and a behavior. The behavior of an object determines how the object will change either spontaneously or as the result of some interaction with its environment. We distinguish:

– Passive objects whose behavior is degenerate and which are under the concurrent control of active objects. Examples of passive objects include all "inert" objects making up the scenery of a virtual environment: chairs, trees etc.
– Active objects endowed with a behavior and which can evolve autonomously within the simulated world. Computer-controlled opponents in games ("bots") are typical examples of active objects.

Note that an active object is not necessarily a "material" visible object. Force fields (gravity, wind etc.) are examples of such objects. However every simulation object has a well-defined spatial extent.

Interactions between simulation objects make the simulation progress over time. In the context of synthetic environments, we define a perception model which quantifies and constraints inter-object interactions. Each simulation object is endowed with an "aura" that describes how far an object can perceive its environment and how far it can influence its environment:

```
class Aura {
  public Bounds getInfluenceZone();
  public Bounds getPerceptionZone();
}
```

A simulation object is endowed with a collection of auras, each dedicated to a specific medium (visual, aural senses etc.).

3 Replication and Persistence Model

The major challenge faced by our platform is to be able to handle large-scale simulation spaces populated with several thousands of simulated objects and accessible in real-time. Centralized architectures which rely on a single server for hosting the simulation space suffer from obvious scalability problems, the server being a natural bottleneck for communication and local processing. The Continuum platform is based on a completely distributed architecture in which a collection of simulations cooperate with one another in order to simulate the conceptually unique simulation space. We distinguish:

- autonomous simulations without the control of human participants. Note that the existence of autonomous simulations guarantees the persistency of (part of) the simulation space,
- human-driven simulations that include functionality necessary to allow human participants to take part into the simulation: graphical rendering, I/O interfaces. It is worth noting that the core of the Continuum platform is not dependent on any graphical/audio rendering APIs and focus only on distributed systems issues.

The task of simulating the large population of objects is spread among these simulations. A given logical simulation object is broken up into a certain (and dynamic) number of replicas. The *SimObject* abstraction is the root class of all simulation objects that can be shared as part of a virtual environment:

```
abstract class SimObject implements EventListener , Serializable {
  public Aura getAura();
  ...
}
```

A *SimObject* represents both:

- a reference to a conceptually unique simulation object which is fragmented over several simulations. In other words, from the outside, one can interact with a *SimObject* instance without being aware of the fact that this instance is a replica.
- an individual replica. Internally a *SimObject* "knows" that it is a replica that needs to synchronize with its peers.

A *SimObject* must provide the way to create replicas of itself. Java serialization is used for this purpose. Any replica must be ready to be passed by value to another simulation in order to create a new replica of the simulation object it represents. Note that, except for immutable objects, this mechanism, by itself, does not solve all consistency issues raised by the replication architecture and needs to be complemented by an object-specific consistency protocol.

At one given instant, a logical simulation object will be replicated on N simulations. Within the group of N replicas that represent this object, we distinguish a *master* replica, for which the platform guarantees the following properties:

- The *master* replica is initially located within the simulation which created and added the object to the virtual environment (i.e. the birthplace of the object)
- a simulation object exists as long as its master replica exists
- at any moment there is one and only one master replica for any simulation object
- the mastership of an object can be transferred from one simulation to another
- the shutdown of a simulation entails the destruction of all simulation objects for which it holds a master copy

This master-slave distinction is quite similar to the notion of proxy in client-server architectures. A slave object acts as a local representative of the master object. How master and slave replicas communicate with one another is covered in the next section. We assume, although this can be qualified to a large extent, that the "intelligence" of the object is located within its master copy whereas the slave replicas are "dumb" and just reflect the state of the master object.

A given simulation hosts a certain number of master replicas (typically those which has been created locally) as well as slave copies of other simulation objects. Instead of replicating the whole simulation space on each simulation, the perception capabilities of objects (as quantified by their auras) are used as criteria to determine the actual subset of the object population which needs to be present locally:

Only objects whose zone of influence intersects the zone of perception of at least one master object need to be replicated locally.

The *SimSpace* abstraction provides access to the simulation space. It is a container of *SimObjects* and allows a client application to add or remove objects to/from the shared environment and to be notified of changes in its population:

```
interface SimSpace {
  void addObject(SimObject obj);
  void removeObject(SimObject obj);
  void addSimSpaceListener( SimSpaceListener listener);
  void removeSimSpaceListener( SimSpaceListener listener);
  void makePersistent(SimObject obj);
}
```

```
interface SimSpaceListener {
   void handleAddObject(SimObject obj);
   void handleRemoveObject(SimObject obj);
}
```

4 Event Model and Synchronization

Replicas which collectively make up a logical simulation object need to keep themselves synchronized in order to provide a consistent local simulation space as hosted by each simulation and, as possibly viewed and manipulated by human users.

A *SimObject* instance interacts with its peers by exchanging events over a many-to-many event channel which is provided by the Continuum platform on a per-object basis:

```
abstract class SimObject implements EventListener , Serializable {
  protected PeerChannel getPeerChannel();
  ...
}
```

The sending of events is performed via the *PeerChannel* interface which abstracts this event channel linking replicas to one another (see figure 1):

```
public interface PeerChannel   {
   public void  sendToAll(Object event, Object hints);
   public void  sendToMaster(Object event, Object hints);
}
```

This interface allows one *SimObject* to emit an event to all other replicas (*sendToAll*) or only to the master replica (*sendToMaster*). The event argument is any serializable Java object.

It is expected that the *PeerChannel* will offer different *sendEvent* semantics. The stronger the semantics, the less work has to been carried out by the *SimObject* proper. The *hints* parameter is currently a placeholder for providing specific requirement on a per-event basis. We expect the following semantics to be offered (the first two have been implemented so far):

- best-effort, no delivery guarantee, no sequenced delivery guarantee
- ordered and reliable delivery with respect to the source
- causal delivery for all events emitted within the interaction group the sender is part of.

Each *SimObject* is a listener for events coming from its peers:

```
interface EventListener {
  public void eventReceived(Event event);
}
```

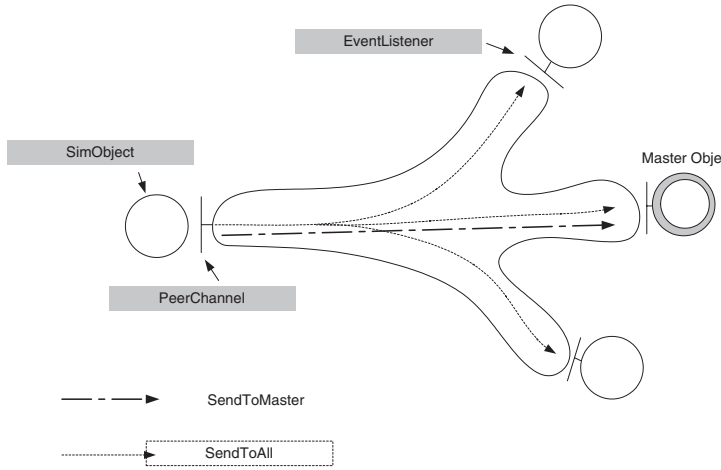


Fig. 1. Inter-replica event communication

Specializations of the *SimObject* class need to implement the *eventReceived* method following a chain of responsibility design pattern [1] along the inheritance tree. In other words, if a class cannot handle an incoming event, it should pass it on to its parent class and so on all the way up to the *SimObject* root class.

An *Event* represents a change in the state of the object. The *Event* abstraction is defined as follows:

```
interface Event {
  long getRealTimeStamp();
  Object getCausalTimeStamp();
  Object getArgument();
}
```

Each *Event* is associated with the wall clock physical time of its emission (*getRealTimeStamp*) and a causal timestamp (*getCausalTimeStamp*) which is currently a placeholder for providing access to the sequence of events causally related to this one (provided that the platform is extended with consistency protocols meeting both real-time and causality ordering constraints between objects).

Consistency Policy Examples

Various types of consistency policies have been implemented on top of the event primitives covered above:

– for highly dynamic mobile objects, the *master* replica "pushes" events containing the kinematic state of the object on a continuous basis. In this case, the loss of one event is not critical since it will be rapidly superseded by later events in the flow. The bandwidth requirement of such event flows is

reduced by employing dead-reckoning techniques. Dead-reckoning is a way to hide network latencies by predicting the kinematic state of an object based on an extrapolation algorithm. Thus the master object will emit an event only if it detects that its real position differs from the approximated position above a certain threshold.
- some attributes of a simulation are static and will never change over the life of the object (e.g. the graphical description of "visible" objects). Such data are usually requested on demand by a slave replica using a "pull" approach.
- concurrency control issued for passive objects are currently handled using a "centralized" approach. All replicas forward the request to alter an object (say to change its color) to the master replica which takes a decision according to a FIFO policy and announces the change back to its replicas.

5 Platform Architecture

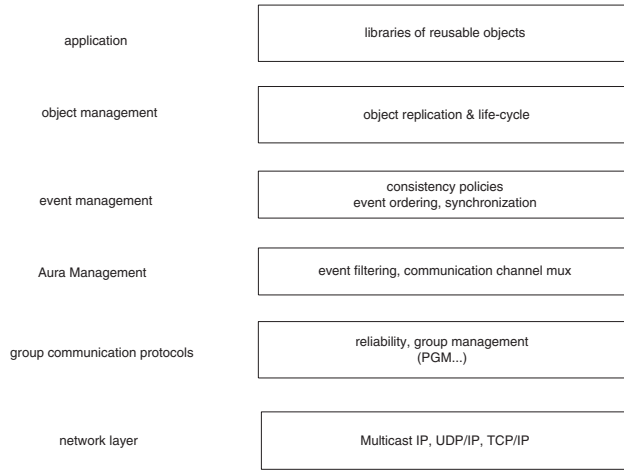


Fig. 2. Continuum platform architecture

The Continuum middleware architecture is depicted in figure 2. The layered architecture reflects a reasonable separation between mostly orthogonal tasks within the system. Worthy of note is the fact that each layer makes a clear separation between policy and mechanisms and can be extended and adapted to fit the needs of the target application.

- The network layer allows the integration of arbitrary network protocols. Of particular interest are multicast protocols well adapted to the communication patterns of shared virtual worlds applications.
- On top of the basic network layer, group communication services are offered which provide various trade-off between reliability and timeliness.

- The Aura Management layer is responsible for filtering the flow of events in accordance with the partial replication model described previously, which relies on object auras. Various types of Aura management implementations are possible. The current implementation relies on a combination of network-level filtering (through a virtual world partition and the allocation of separate IP multicast addresses for each region) and local filtering.
- The Event Management layer provides various levels of consistency and involves tasks such as ordering, timestamping or buffering events.
- The object management layer covers the correct handling of replicated objects (bookkeeping, lifecycle, garbage collection).
- The application layer consists of simulation objects containing specific information such as graphical models, behavior, rules of interaction between entities (e.g. physical simulation). The Continuum platform expects to offer along with the software framework a toolkit of reusable components from which an application programmer can construct complex objects.

The Continuum platform relies on the open distributed object platform Jonathan [4]. Jonathan offers both a CORBA-compliant and a RMI-like programming personalities. The client-server functionality of the RMI personality has been used for most point-to-point communications in Continuum (initial access and connection to a world server etc.). In the context of Continuum, this ORB has been extended with many-to-many event propagation facilities relying on IP multicast.

6 Related Work

Several platforms for large-scale shared virtual environments have been developed in the past years [6,10,9,2]. These systems are based on a shared database approach and focus on the replication of passive objects. They assume that the behavior associated with objects is handled outside the scope of the replication platform proper. Among them, very few can qualify as open middleware platforms. They usually make fixed design choices in terms of communication architecture, consistency protocols etc. For example, using a reliable multicast protocol for event dissemination on a systematic basis might not be appropriate for all types of events. As a result, their extensibility is limited and they are biased toward certain types of applications at the expense of others.

A lot of work has been done in the areas of replication and cache consistency. Group communication systems relying on strong consistency criteria [13], standard distributed database or shared memory approaches [11] are ill-suited for real-time interaction and will scale only if one accepts a performance penalty or a significant delay. Resorting to partial replication in order to reduce the degree of sharing is an approach adopted by many systems in the field of collaborative work. The notion of aura is introduced in [7] as a spatial notion to support interaction models. This notion is further refined in [8] to quantify the mutual awareness between two objects. Other approaches [14,9] rely on a static partitioning of the shared information space into a set of disjoint interaction spaces.

The design of real-time consistency criteria and protocols is still an open research area. Resorting to application-level prediction techniques such as dead-reckoning is an approach adopted by many virtual environment platforms [12,10].

7 Conclusion

Distributed multi-user applications on the Internet (and shared 3D worlds in particular) require an underlying infrastructure which takes into account their specific needs and which can adapt to their specific semantics. By designing the Continuum middleware as an open software framework with a clear-cut separation between policies and mechanism, we offer an adaptable and flexible platform which both (i) insulates the application programmer from low-level aspects and (ii) enables the system programmer to introduce alternate infrastructure components (network protocols, consistency protocols etc.). The use of the Java programming language, beyond its cross-platform portability, significantly contributes to the flexibility of the platform with the ability to dynamically download and run code. Several test applications have been developed and experimented over a WAN using IP multicast. In particular an underwater world simulation involving several hundred of autonomous fishes has been developed. The computation of the behavior of the fishes is spread across several simulations. Thanks to aura-directed event filtering, a given user only "sees" a small subset of the simulation space at a given time, which significantly decreases the bandwidth requirement of the network link connecting his client simulation.

We are currently assessing the performance of our platform in more detail. We intend to extend it toward the following directions:

- integration of our platform with the Java-based reactive programming framework described in [5]. This approach avoids the use of Java threads and the not-clear semantics and non deterministic scheduling associated with them.
- real-time consistency protocols based on consistency criteria which take into account both real-time and causality relationships between events.
- scalability issues need to be investigated further regarding, e.g., placement, caching and load-balancing issues.

References

1. Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John: "Design Patterns, Elements of Reusable Object-Oriented Software", Addision-Wesley, published October 1994.
2. High Level Architecture Web Site: http://hla.dmso.mil/
3. Floyd S., Jacobson V., Liu C., McCanne S., Zhang L. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing", Proc. ACM SIGCOMM'95, 1995.
4. Dang Tran F., Dumant B. , Horn F. , Stefani J.B., "Jonathan: an open distributed processing environment in Java", Middleware'98, IFIP International Conference on distributed Systems Platforms and Open Distributed Processing, Lake District, UK, September 1998. Jonathan Web site: http://www.objectweb.org

5. Hazard L., Susini J.F., Boussinot F., "Distributed Reactive Machines", RTCSA'98, Hiroshima, October 98.
6. Hagsand O., "Interactive Multiuser VEs in the DIVE System", IEEE Multimedia, Vol.3, Number 1, 1996.
7. Benford S., Fahlen L., Greenhalge C., Bowers J., "Managing mutual awareness in collaborative virtual environments", Proceedings of the ACM SIGCHI conference on Virtual Reality and Technology, August 1994, Singapore, ACM press.
8. Greenhalgh C., Benford S., "MASSIVE: a Distributed Virtual Reality System Incorporating Spatial Trading", Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, Canada, May 30 - June 2, 1995, IEEE Computer Society.
9. Anderson D., Barrus J., Howard J., Rich C., Shen C., Waters R., "Building Multi-User Interactive Multimedia Environments in MERL", IEEE Multimedia, Volume 2, Number 4, 1995.
10. Macedonia M., Zyda M., Pratt D., Barham P., Zeswitz S., "NPSNET: A Network Software Architecture for Large Scale Virtual Environments", Presence, Volume 3, Number 4, 1990.
11. Li K., Hudak P., "Memory Coherence in shared virtual memory systems", ACM Transactions on Computer Systems 7(4), 1989.
12. Roberts D., Sharkey P., "Maximizing concurrency and scalability in a consistent, causal, distributed virtual reality system, whilst minimizing the effect of network delays", IEEE Sixth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, MIT, June 1997.
13. Birman K., Schiper A., Stephenson P., "Lightweight causal and atomic group multicast", ACM Trans. on Computer Systems, Vol. 9 no 3, 1991.
14. Kindberg T., Coulouris G., Dollimore J., Heikkinen Jyrki, " Sharing Objects over the Internet: the Mushroom Approach", Proceedings of Global Internet 96, IEEE, London, November 1996.

TACO — Dynamic Distributed Collections with Templates and Topologies

Jörg Nolte, Mitsuhisa Sato, and Yutaka Ishikawa

Real World Computing Partnership
Tsukuba Mitsui Bldg. 16F, 1-6-1 Takezono
Tsukuba 305-0032, Ibaraki, Japan
{jon,msato,ishikawa}@trc.rwcp.or.jp

Abstract. High-level data-parallel programming with distributed object sets eases many aspects of parallel programming. In this paper we describe the design and implementation of TACO, a template library that provides higher-order operations on polymorphic regular and irregular distributed object sets by means of reusable topology classes and C++ function templates.
Keywords: dynamic collections, distributed data-parallel processing, global communication

1 Introduction

Collective operations are a powerful means to implement globally coordinated operations on distributed data sets. Despite this fact the acceptance of data-parallel programming languages is still limited. However, while efficient fine-grained data-parallel computing is hard to achieve without proper compiler support and appropriate optimization techniques, efficient coarse-grained as well as medium-grained data-parallel computing is well possible on a library basis only. Thus we can introduce the flavour of data-parallel programming without language extensions and data-parallel programming can be conveniently combined with common parallel programming techniques where appropriate. In addition, the inheritance mechanisms of object oriented languages allow us to control, extend and adapt the behaviour of distributed collections in a well structured manner.

TACO is therefore a pure template library that extends our basic Multiple Threads Template Library (MTTL) [8] with **T***opologies* **a***nd* **Co***llections*. Topology classes are used to describe the *relation* between members of distributed object groups and template classes as well as function templates are applied to address such data sets collectively.

2 The Multiple Threads Template Library

The MTTL is a very efficient communication and threading library on top of either the PM [9] or MPI communication layer. Amongst several other features

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1071–1080, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the MTTL provides a concept for *global pointers*, multi threaded remote method invocation and global synchronization variables.

Global pointer templates store both the network address as well as the local address of an object, such that the object can be read and written remotely using overloaded operators. Objects can be created remotely resulting in a global pointer that can also be applied for remote method invocation. Remote method invocation is implemented by means of function templates that take care of argument marshaling, communication and method invocation. A method can either be invoked synchronously or asynchronously. In case of asynchronous invocations, global pointer based synchronization variables can be used for lazy synchronization. A synchronization variable is a kind of future [7], that blocks any reading access until the variable is (remotely) initialized. These few prerequisites were already sufficient to implement distributed collections and efficient higher-order operations.

2.1 Global Object Pointers and Remote Method Invocation

The original implementation of the MTTL did not support polymorphism. In the MTTL a global pointer to a derived class cannot be assigned or safely converted to a pointer to a base class like in standard C++. Likewise a method belonging to a base class cannot be invoked using a global pointer to a derived class. Therefore we first extended the MTTL with a very lean remote method invocation layer (RMI) that implements fully polymorphic global pointers and remote method invocation mechanisms.

We designed an `ObjectPtr` class for the MTTL that implements statically type safe compatibility between global object pointers and remote method invocation according to C++ type rules.

```
class Base {
  public:
    virtual void f(int n);
};

class Derived : public Base {
  public:
    Derived(int i);
    void f(int n);
};

// allocate new object at node 17 and pass 77 as argument to
// the constructor
ObjectPtr<Derived> derived = allocate<Derived> (17) (77)
ObjectPtr<Base> base;

base = derived;             // OK, statically checked.
base->call(&Base::f, 77);   // remote Derived::f(77) is called.
```

Note that `&Base::f` does not evaluate to the machine address of a method as one might expect but to an index into the virtual table of class `Base` because

`f()` is declared to be a virtual method in `Base`. The various invocation methods of the `ObjectPtr` class take a pointer to the member function to be invoked as well as an arbitrary number of arguments being passed to the method. All methods are implemented as various overloaded member function templates being distinguished automatically by the compiler by means of the actual type of the passed arguments. Parameter passing semantics is strictly call-by-value and global object pointers can be passed as well.

```
ObjectPtr<Class> ptr = allocate<Class> (where)(arglist);
Result result = ptr->call(&Class::method,arglist);
ptr->apply(&Class::void_method,arglist);
ptr->apply(Sync<Result>,&Class::method,arglist);
```

Remote objects are created by means of the `allocate()` template at the specified node, the specified arguments are passed to the constructor of `Class`. The `call` methods are synchronous, while the `apply()` methods are asynchronous. The most simple variant of `apply()` is only applicable to methods returning `void` while the second version of `apply()` allows for lazy synchronization of the result of the method. This is particularly useful for the implementation of global reductions.

The implementation of class `ObjectPtr` is based on the MTTL's monomorphic `GlobalPtr` class. The compatibility of global object pointers is defined by a conversion constructor:

```
template<class T>
class ObjectPtr: public GlobalPtr<T> {
public:
   // retrieve the local pointer component
   T* localPtr() const { return (T*)getLaddr(); }
   // retrieve the logical network address
   int location() const { return getPe(); }

   template<class Derived>
   ObjectPtr(const ObjectPtr<Derived>& other)
   {
       // legal if "T" is a base of "Derived",
       // or "Derived*" is compatible to  "T*"
       T* p = other.localPtr();
       // now initialize GlobalPtr base class
       set(p, other.location());
   }
   ...
};
```

Any time an `ObjectPtr` instance is used in a context that expects a different type of global object pointer this conversion constructor is called automatically to perform the type coercion. The global object pointer is effectively split into a local pointer component and node address which are now both assigned separately. Thus a type coercion on the local pointer component is automatically performed according to C++ type rules. The coercion fails when the types of

the local pointer components are not compatible to each other and these errors are detected at compile time.

For performance reasons we also provide an overloaded assignment operator, that basically performs the same operations as the conversion constructor. This operator helps to avoid the creation of temporary objects and additional copies in assignments. The `ObjectPtr` class is now powerful enough to support various kinds of dynamic as well as polymorphic distributed data-structures.

3 Collections and Topologies

In TACO the structure of a collection is defined by a topology class. In principle, a collection can have any user defined topology such as n-dimensional grids, lists or trees (fig. 1).

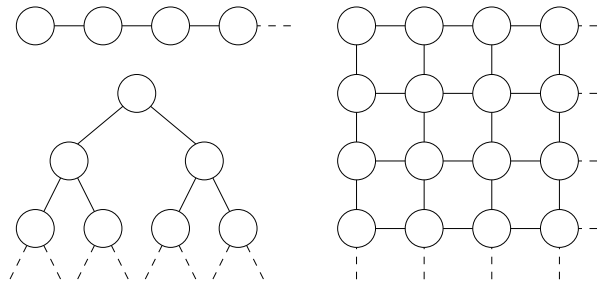


Fig. 1. Topologies

Topologies are implemented as distributed linked object sets by means of global pointers to objects (refer to section 2) These sets can be collectively addressed by issuing a method to a *leader object* that represents the entire collection. All objects that are reachable from this leader object automatically participate in collective operations.

Note that multiple objects of a collection might be mapped to the same computing node. TACO will apply local invocation mechanisms in this case. Thus the size of collections is not limited by the number of available nodes.

3.1 Collective Methods

We implemented only those collective operations that are fundamental to most collective operation patterns. An asynchronous global `map()` operation is provided to initiate data parallel computations and invoke a `void` method on all members of the collection. The asynchronous `map()` operation is complemented with a synchronous `reduce()` operation that executes a method on all members of the collection in parallel and applies an associative and commutative binary

function to combine all results. Collective methods are again implemented as member function templates of the `ObjectPtr` class.

```
ObjectPtr<Class> ptr = ...;
ptr->map(&Class::method,arglist);
Result res = ptr->reduce(&red,&Class::method,arglist);
```

All collective methods assume that the members of a collection define an STL style iterator [12] that (recursively) iterates all members of a collection. Members are identified uniquely by means of *global object pointers* (see section 2) and thus the iterators need to evaluate to global object pointers. TACO's topology classes define such iterators and thus application classes only need to be derived from a topology class to inherit the iterators as well.

The iterator determines the order in which collective methods are passed to members of the collection. Each collective method is first propagated asynchronously to the other members of the collection in the order defined by the iterator before it is called on the local object instance. In case of reductions we use the asynchronous `apply()` method in conjunction with lazy synchronization to synchronize on the availability of results. All results are collected in exactly the same order in which the invocation method has been spread. Thus, the behaviour remains deterministic and application programmers can get control over collective operations either through the iterator method of the topology class or by means of custom iterators defined by application classes.

3.2 Creation of Collections

Collections with a known initial size can be created and addressed by means of a `GroupOf` template class as follows:

```
class Sheep: public BinTree<Sheep> {
 public:
  int weight(); // return sheep's weight
  ...
};

Cyclic map(8); // cyclic mapping to 8 nodes
GroupOf<Sheep> flock(128, map); // 128 members

// a reduction (global sum)
int add(int a, int b) { return a+b; }
int sum = flock.reduce(&add, &Sheep::weight);
```

In this example a collection of 128 `Sheep` is created and mapped to 8 processing nodes using a `Cyclic` mapping strategy. Block cyclic as well as random strategies are also available. Mapping classes provide a subscript operator that delivers for a given rank in a collection a corresponding node number. The mapping argument of the `GroupOf` class is generic and thus even an array of node numbers can be specified to control mapping explicitly. Class `Sheep` can be any arbitrary C++ class, while `BinTree` is a topology class that describes a distributed binary tree

of objects of any kind. After the construction we can sum up the weight of the whole flock by means of a global reduction that applies an `add()` function to reduce the results.

3.3 Design Considerations

In the previous example class `Sheep` is aware that its instances will become members of a collection being organized as a binary tree. Being aware of collections and topologies is beneficial to those classes that need to determine topology related data at runtime, such as the rank of objects within the topology, the addresses of neighbouring objects in case of grid topologies and similar data. This approach is intrusive and thus this collection concept cannot directly be applied to arbitrary classes.

However, collection aware classes can easily be separated from other classes by means of inheritance. Therefore a `Sheep` class that is aware of being in a flock can easily be described be means of multiple inheritance:

```
class FlockMember: public Sheep, public BinTree<FlockMember>
{
   ...
};
```

The `FlockMember` class inherits all abilities from the `Sheep` class and additionally knows about its group relationship. The methods of `FlockMember` can now easily deal with conditional collective operations without any interference with the basic `Sheep` class.

We automated this approach and provide additionally a non-intrusive `CollectionOf` template that does not require its members to be derived from a topology class. The topology is specified to the `CollectionOf` template instead and members are wrapped into containers that are derived from the specified topology class.

```
CollectionOf<Sheep,BinTree> flock(...);
```

Note that collective operations are applied in the same manner to both intrusive and non-intrusive collections, only the declarations of the collections as such differ. Furthermore, constructors of collections might have an arbitrary number of arguments that are passed to the members for initialization. When the scope of a collection object becomes invalid (or the collection object is deleted in case it has been allocated on the heap) all members of the distributed collection will also automatically be destroyed.

3.4 Dynamic Collections

The implementation of dynamic collections is a straightforward task since TACO's collections are inherently based on distributed dynamic data structures that can be constructed in the same way as any corresponding local data structure. Therefore the implementation of highly dynamic schemes is often possible with a few lines of code only.

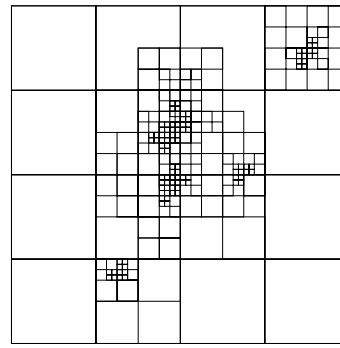


Fig. 2. An adaptive Grid

All TACO topology classes provide a `join()` method that allows to add either new members to the collection or to join existing collections. To illustrate this further, consider a class `Quadrant` that adaptively refines a mesh or grid as depicted in figure 2 based on some runtime condition. The following short code fragment then describes the dynamic construction of the corresponding collection:

```
class Quadrant: public QuadTree<Quadrant> {
   int x, y, length;
public:
   ...
   int  count() { return 1;}
   void refine()
   {
      // evaluate some condition for refinement
      if (isLeaf() && ...) {
         int len = length/2;

         // create and join new members ...
         join(allocate<Quadrant>(Random()) (x, y, len));
         join(allocate<Quadrant>(Random()) (x+len, y, len));
         join(allocate<Quadrant>(Random()) (x, y+len, len));
         join(allocate<Quadrant>(Random()) (x+len, y+len, len));
      }
   }
};
```

The `QuadTree` class is a 4-ary tree topology. Depending on the state of the object and/or some specific runtime condition the `refine()` method may create new members and join them to the tree. The `allocate()` template is applied to create a new remote `Quadrant` instance according to a random placement strategy and returns a global object pointer to the newly created object.

When `refine()` is now called as a collective operation the members of the collection that meet the condition for refinement will create new members in parallel and join them to the collection dynamically.

```
// allocate first group member
ObjectPtr<Quadrant> coll = allocate<Quadrant>(random()) (0, 0, 4096);

for (...) // intiate several parallel refinements
   coll->map(&Quadrant::refine);

// count actual members using a  global reduction
int add(int a, int b) { return a+b; }
int number = coll->reduce(&add, &Quadrant::count);
```

By means of such schemes highly irregular collections as shown in figure 2 can be created. Since we also retain full polymorphism, these dynamic collections can be heterogenous as well.

4 Performance

We measured the performance of our collective operations on the RWC PC Cluster II consisting of 128 Pentium Pro nodes (200MHz CPU) interconnected by a 160MByte/sec Myrinet network running the Score [9] system software on top of a Linux System. All benchmarks have been run 100 times in a loop to

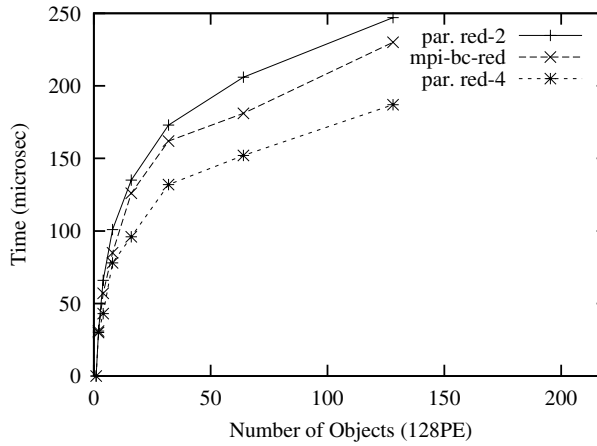


Fig. 3. reduce() vs. MPICH-PM/CLUMP

ensure, that the benchmark code was mostly in the cache and that the benchmark times were long enough to get a meaningful measurement using the standard

`gettimeofday()` system call. The resulting times have been divided then by the number of runs.

For comparison with a standard communication library we measured the computation of global sums with our reductions against collective operations of MPI (fig. 3) using both a binary tree (par. red-2) as well as a 4-ary tree (par. red-4) to study the impact of different tree topologies (which are both not optimal for software multicasts [6]). A `MPI_bcast()` followed by a `MPI_Reduce()` (mpi-bc-red) is closest to our `reduce()` semantics. The MPI implementation we used is MPICH-PM/CLUMP [13] which is a very efficient port of MPICH for Myrinet networks. Figure 3 shows that even our very first implementation is by all means in the competitive range. In case of the 4-ary tree topology our implementation shows even better performance than the corresponding MPI implementation.

5 Conclusion

Collections and collective operations are neither new nor did we invent the notion of topologies as such. In fact our work has been strongly inspired by the collection concept of pC++ [3] and ICC++ [5], the *communities* in Ocore [10], *groups* in HPC++ [1] and – last but not least – the topology concept of Promoter [2].

However, in most data-parallel languages as well as commonly used communication libraries like MPI the notion of a group is usually an abstract concept and the programmer has to rely on an efficient implementation of this concept without any reasonable means of control. Since it is notoriously hard to find suitable implementations of collections that perform well on all use cases and all network architectures, TACO gives the programmer a powerful means of control over collections through reusable *topology* classes, that reveal the internal structure of a collection. This is of utmost importance with regard to performance tuning.

Furthermore, most data-parallel approaches traditionally concentrate strongly on regular array structures like the Amelia Vector Template Library [11]. Amelia is a single-paradigm approach that solely relies on data parallelism similar to a vector machine. The only supported data structure is a distributed vector of potentially very fine-grained elementary data types like doubles or integers. Because of this fine-grained library approach Amelia needs to provide a rich set of specific algorithms on vectors to achieve acceptable performance.

Compared to Amelia, TACO is a fairly general purpose library for medium grained parallelism. TACO focuses on a simple, yet powerful and easy to understand group concept that allows programmers to construct more complex application specific libraries with modest effort. We deliberately based our collections on flexible graphs similar to ARTS [4]. Therefore existing collections can easily be changed dynamically at run time and the implementation of adaptive schemes is straightforward. Due to TACO's full support for polymorphism heterogenous collections are possible as well.

Although fine grained parallel computing is also possible, we do not expect satisfactory performance results since we cannot apply compiler level optimiza-

tions like the language-based approaches. However, the performance for fine grained structures can be significantly improved when the members of TACO's collections are themselves containers holding entire sets of small objects. Therefore TACO can serve well as a basis for domain specific libraries as well as runtime systems for fine-grained data-parallel processing.

References

[1] Peter Beckman, Dennis Gannon, and Elizabeth Johnson. Portable Parallel Programming in HPC++. Technical report, Department of Computer Science, Indiana University , Bloomington, IN 47401.

[2] M. Besch, H. Bi, P. Enskonatus, G. Heber, and M. Wilhelmi. High-Level Data Parallel Programming in PROMOTER. In *Proc. Second International Workschop on High-level Parallel Programming Models and Supportive Environments HIPS'97*, Geneva, Switzerland, April 1997. IEEE-CS Press.

[3] Francois Bordin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), Fall 1993.

[4] Lars Büttner, Jörg Nolte, and Wolfgang Schröder-Preikschat. ARTS of PEACE – A High-Performance Middleware Layer for Parallel and Distributed Computing. *Journal of Parallel and Distributed Computing*, 59(2):155–179, Nov 1999.

[5] A. Chien, U.S. Reddy, J.Plevyak, and J. Dolby. ICC++ – A C++ Dialect for High Performance Parallel Computing. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software, ISOTAS'96*, Kanazawa, Japan, March 1996. Springer.

[6] J. Cordsen, H. W. Pohl, and W. Schröder-Preikschat. Performance Considerations in Software Multicasts. In *Proceedings of the 11th ACM International Conference on Supercomputing (ICS '97)*, pages 213–220. ACM Inc., July 1997.

[7] R. H. Jr. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.

[8] Y. Ishikawa. Multiple threads template library. Technical Report TR-96-012, Real World Computing Partnership, 1996.

[9] Yutaka Ishikawa, Hiroshi Tezuka, Atsuhi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O'Carroll, and Hiroshi Harada. RWC PC Cluster II and SCore Cluster System Software – High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pages 55 – 62, 1999.

[10] H. Konaka, M. Maeda, Y. Ishikawa, T. Tomokiyo, and A. Hori. Community in Massively Parallel Object-based Language Ocore. In *Proc. Intl. EUROSIM Conf. Massively Parallel Processing Applications and Development*, pages 305–312. Elsevier Science B.V., 1994.

[11] Thomas J. Sheffler. The Amelia Vector Template Library. In *Parallel Programming using C++*, pages 43–90. MIT Press, 1996.

[12] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34, Hewlett Packard Laboratories, 1994 revised 1995.

[13] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and Evaluation of MPI on an SMP Cluster. In *Parallel and Distributed Processing – IPPS/SPDP'99 Workshops*, volume 1586 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1999.

Object-Oriented Message-Passing with TPO++*

Tobias Grundmann, Marcus Ritt, and Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik, University of Tübingen
Department of Computer Engineering, Sand 13, 72076 Tübingen
{grundman,ritt,rosen}@informatik.uni-tuebingen.de

Abstract. Message-passing is a well known approach for parallelizing programs. The widely used standard MPI (Message passing interface) also defines C++ bindings. Nevertheless, there is a lack of integration of object-oriented concepts. In this paper, we describe our design of TPO++[1], an object-oriented message-passing library written in C++ on top of MPI. Its key features are easy transmission of objects, type-safety, MPI-conformity and integration of the C++ Standard Template Library.

1 Motivation and Design Goals

With MPI, a widely accepted standard for message-passing has been established. At the same time, object-oriented programming concepts gain increased acceptance in scientific computing (see, for example [1]).
The MPI-2 standard [6, 7] defines C++ language bindings for MPI-1 and MPI-2, but these bindings are no significant improvement compared to the C bindings. The interface is not type-safe, does not simplify the MPI calls and defines no way for transmitting objects. Other approaches such as the mpi++ system [4, 5], OOMPI [8] and Para++ [2] improve certain aspects of message-passing in C++, but, besides some other topics, none of them integrates the STL (Standard Template Library), an important part of the current C++ standard [3]. Further they often don't support user-defined types very well or there is a significant difference to MPI conventions, even if not necessary to introduce object-oriented concepts. In the design of TPO++, we try to address these problems:
A major goal in providing a class library for C++ is a tight integration of object-oriented concepts, in particular the capability of transmitting objects in a simple, efficient, and type-safe manner. Also, an implementation in C++ should take into account all recent C++ features, like the usage of exceptions for error handling and the integration of the Standard Template Library by supporting the STL containers as well as adopting the STL interface conventions. The interface design should conform to the MPI conventions and semantics as closely as possible without violating object-oriented concepts. This helps migrating from C bindings and eases porting of existing C or C++ code. Further, the implementation should not differ much from MPI in terms of communication and memory

* This project is funded by the DFG within SFB 382.
[1] Tübingen Parallel Objects

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1081–1084, 2000.
© Springer-Verlag Berlin Heidelberg 2000

efficiency. Another goal is to guarantee thread-safety to provide maximum flexibility for application software and parallel runtime systems. This topic will not be further discussed here since thread-safety is optional in the MPI-Standard and depends mostly on the underlying MPI-Implementation.

2 Interface and Examples

The basic structure given in the C++ bindings of MPI is similar to our approach. All common MPI objects, i.e. communicators, groups, are implemented as separate classes. After initialization, the user can use the global Communicator object `CommWorld`.

Transmission of predefined C++ types In the case of sending a C++ basic type, the send call reduces to:

```
double d;
CommWorld.send(d, dest_rank, message_tag);
```

STL containers can be sent using the same overloaded communicator method. The STL conventions require two iterators specifying begin and end of a range, which also allows to send subranges of containers:

```
vector<double> vd;
CommWorld.send(vd.begin(), vd.end(), dest_rank, message_tag);
```

The application can also use the blocking, synchronous and ready send modes defined in MPI by calling the communicator methods `bsend`, `ssend` and `rsend`, respectively. Asynchronous communication methods return an object of class `Request` which can be used for the application to test or wait for the completion. On the receiver side, a receive-call is done as follows (basic type):

```
Status status;
status=CommWorld.recv(d);
```

Note that the status object, different from MPI, is a return parameter, because error handling is done via exceptions. This simplifies the receiver code, if no error checking is necessary and makes send and receive calls more symmetric. The receive methods take two optional arguments, the senders rank and a message tag for selecting particular messages. If omitted, they default to any sender and any tag, respectively.
To receive a container, a single argument, specifying the insertion point is sufficient. Conforming to the STL, the data can be received into a container which is large enough by means of an iterator, or into any container by means of an *inserter*. The example shows both approaches:

```
vector<double> vd1(x);  // must provide enough space
vector<double> vd2;
CommWorld.recv(vd1.begin());
CommWorld.recv(tpo_back_insert_iterator(vd2)); //allocates space
```

Transmission of user-defined types To enable a class for transmission, the user has to declare its *marshalling category*. The library distinguishes user-defined objects having a trivial copy constructor and complex user-defined objects. Enabling a class with a trivial copy constructor for transmission reduces to the statement `TPO_TRIVIAL(User_type)`. On transmission, the memory block occupied by such an object will be copied directly to the net.
For transmitting complex objects (i.e. without a trivial copy constructor), the user has to define the marshalling methods `serialize` and `deserialize`, as part of the class definition. The presence of these methods must be signaled by a declaration of `TPO_MARSHALL(User_type)`. They are supplied with a serializer object. In a serialize method, `insert()` is called repeatedly for every member to prepare the object for transmission. The serializer object does not copy the data, but records its memory layout for later transmission. Similarly, a received message can be unpacked to user-provided memory locations by calling `extract()` in the deserialize method.
Usually, user-defined objects do not have to inherit from any special "message" class. Also note, that the code given in the marshalling methods can be reused in derived classes.
For applications relying on virtual methods and generic interfaces an abstract base class `Message` is also provided.

3 Comparison with MPI

The tests have been done on Sun Ultra 5 machines (Solaris 2.7) using MPICH 1.1.2 and on a Cray T3E a Cray T3E (512 nodes at 450 MHz) using the native MPI implementation. We measured the efficiency of our library using a ping test and compared the achieved latencies and bandwidths of MPI and TPO.

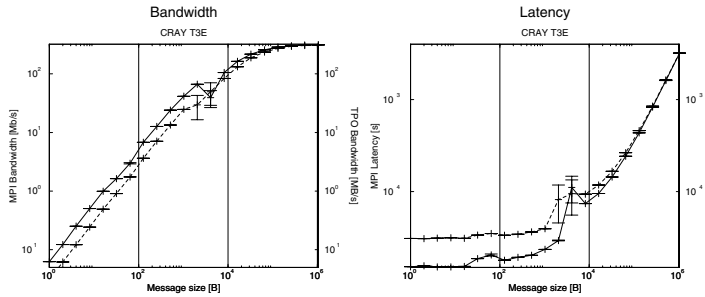


Fig. 1. Comparison of MPI (solid curves) and TPO (dashed curves) on a Cray T3E.

As shown in Figure 1 communication using TPO achieves the same bandwidth as MPI for messages larger than approximately 16 KB. The loss in bandwidth

below this size is mainly due to the increased latency. Latencies of MPI and TPO converge as messages are getting larger. For small messages TPO shows a constant latency overhead of $15\mu s$ compared to MPI.

4 Conclusions

We have presented our implementation of an object-oriented message-passing system. It exploits object-oriented and generic programming concepts, allows the easy transmission of objects and makes use of advanced C++ techniques and features as well as supporting these features, most notably it supports STL datatypes. The system introduces object-oriented techniques and type-safety to message-passing while preserving MPI semantics and naming conventions as far as possible. This simplifies the transition from existing code. In contrast to other implementations the code to marshall an object can be reused in derived classes. Also, our library is able to handle arbitrary complex and dynamic datatypes. An object-oriented interface can be implemented with almost identical performance compared to MPI. The library is designed as a base for parallelizing scientific applications in an object-oriented environment.

References

[1] F. Bassetti, K. Davis, and B. Mohr, editors. *Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'99), European Conference on Object-Oriented Programming (ECOOP'99)*, Technical Report FZJ-ZAM-IB-9906. Forschungszentrum Jülich, Germany, June 1999.

[2] O. Coulaud and E. Dillon. Para++: C++ bindings for message-passing libraries. Users guide. Technical report, INRIA, 1995.

[3] International Standards Organization. Programming languages – C++. ISO/IEC publication 14882:1998, 1998.

[4] D. Kafure and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995.

[5] D. Kafure and L. Huang. Collective communication and communicators in mpi++. Technical report, Department of Computer Science Virginia Tech, 1996.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.

[7] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[8] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of the POOMA conference*, 1996.

Topic 17
Architectures and Algorithms for Multimedia Applications

Manfred Schimmler

Local Chair

The emergence of multimedia technology in recent years is strongly driven by an enormous commercial potential. For the scientific community this development is interesting because a number of attractive disciplines for computer science and engineering flow together into the multimedia mainstream: image processing, computer graphics, data compression, encoding, cryptography, and broadband communication, to mention just a few of them. These fields have always been driving forces behind the design of massively parallel architectures and algorithms as well as special purpose processors and storage systems.

The area provides an additional challenge due to the fact that the time interval from the scientific idea to the implementation becomes shorter and shorter. Take the following example as an indication of this statement on the one hand, and, on the other hand, as a proof of the high quality of this conference series: In the sessions [1, 2] corresponding to this one of Euro-Par'98, and Euro-Par'99 we have seen the presentation of ideas that have meanwhile found their way into successful commercial products.

For this topic four papers in this tempting area have been selected: The first one gives a new approach to the design of array processors for DCT algorithms as used in MPEG-style encoders or decoders. The second paper proposes a massively parallel combination of SIMD array and ISA for volume rendering. An automated approach to design an architecture for image processing algorithms is discussed in the third paper. The last article of this topic provides a parallel storage architecture capable to deliver reliable video data while supporting a maximum number of video streams even in case of failure.

In contrast to the earlier papers on multimedia focussing on special purpose architectures, one may notice that the systems presented in this topic are designed for flexibility in order to cope with future standards. It will be interesting to observe if this trend will hold on in the future.

References

[1] Workshop 10+17+21+22: Theory and Algorithms for Parallel Computation. Proc. Euro-Par'98, LNCS 1470, Springer Verlag, pp. 863-966 (1998)

[2] Topic 12: Architectures and Algorithms for Vision and other Senses. Proc. Euro-Par'99, LNCS 1685, Springer Verlag, pp. 939-1018 (1999)

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1085–1085, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Design of Multi-dimensional DCT Array Processors for Video Applications

Shietung Peng[1] and Stanislav Sedukhin[2]

[1] Hosei University,Tokyo 184-8584, Japan,
speng@k.hosei.ac.jp
[2] The University of Aizu, Aizu-Wakamatsu City, Fukushima 965-8580, Japan
sedukhin@u-aizu.ac.jp

Abstract. In this paper, we propose array processors for computing the 2-D and 3-D DCTs. We first introduce a new method, called dimensional splitting method, for the design of array processors for multi-dimensional image transforms. The method can be applied to any multi-dimensional image transforms with separable kernels such as DFT and DCT. Then, we propose a new coding scheme for the 1-D DCT in which the need for generating the kernel matrix in advance is eliminated. Finally, we show array processors for computing r-D DCT ($r \geq 2$) which are scalable, regular, locally-connected, and fully-pipelined.

1 Introduction

Recent advances in various aspects of digital technology have made possible many applications of digital video such as HDTV, teleconference, and multimedia communications. These applications require high-speed transmission of vast amounts of video data. Most video standards use discrete cosine transform (DCT) as a standard transform coding scheme [4,7]. The DCT is very computationally intensive. To realize high-speed and cost-effective DCT for video coding, one needs a specially designated array processor so that the high throughout requirements can be matched. There has been considerable research in mapping efficient algorithms to practical and feasible VLSI implementations in the recent past [2,5,6]. However, these designs employed irregular butterfly structures with global communications resulting in complex layout, timing, and reliability concerns which severely limit the operating speed and expandability in VLSI implementation.

In this paper, we propose new array processors for computing multi-dimensional DCT/IDCT based on a new decomposition method for image transforms. The main feature of this method is that no transposition of any intermediate matrix is needed during processing. Moreover, our method can be applied to any r-D ($r \geq 2$) image transform. Using this method, the design of the array processors for the r-D DCT depends heavily on the design of corresponding linear array processors for the 1-D DCT. We develop effective DCT coding schemes for the 1-D DCT. Our schemes eliminates the need for generating the whole kernel

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1086–1094, 2000.
© Springer-Verlag Berlin Heidelberg 2000

matrix of the 1-D DCT. The coefficients are created locally on the fly whenever needed. This I/O effective recursive scheme is of particular interests when the real-time systems such as HDTV are considered [1,3].

2 A Dimensional Splitting Method

The 1-D image transforms can be expressed in the terms of the following relation

$$T(u) = \sum_{x=0}^{N-1} f(x)g(x,u),$$

where $0 \leq u \leq N-1$, $T(u)$ is the transform of $f(x)$, and $g(x,u)$ is the transformation kernel. The properties of the transformation kernel determine the nature of a transform. Similarly, the 2-D and the 3-D image transforms can be expressed as

$$T(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y)g(x,y,u,v),$$

and

$$T(u,v,w) = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \sum_{z=0}^{N-1} f(x,y,z)g(x,y,z,u,v,w),$$

where $0 \leq u,v,w \leq N-1$, $g(x,y,u,v)$ and $g(x,y,z,u,v,w)$ are the kernels for the 2-D and the 3-D transforms, respectively. A 2-D kernel is said to be separable if $g(x,y,u,v) = g_1(x,u)g_2(y,v)$, and it is symmetric if g_1 is functionally equal to g_2. The separability and symmetry of a 3-D kernel are defined similarly. Some samples of transforms with separable, symmetric kernels are discrete Fourier transform, discrete cosine transform, Walsh-Hadamard transform, and Haar transform.

A 2-D (3-D) transform with separable kernel can be computed in two (three) steps, each requiring a 1-D transform. In the 2-D case, we can first take the 1-D transform along each row of $f(x,y)$ to get $z(u,y)$, and then, take the 1-D transform along each column of $z(u,y)$ to get the result $T(u,v)$. That is,

$$T(u,v) = \sum_{y=0}^{N-1} (\sum_{x=0}^{N-1} f(x,y)g_1(x,u))g_2(y,v) = \sum_{y=0}^{N-1} z(u,y)g_2(y,v).$$

Similarly, in the 3-D case, the 1-D transforms are performed three times, each on different dimension, to get the result $T(u,v,w)$.

The idea of the proposed method to construct r-D array processors for computing the r-D image transform with separable kernel is based on the above approach. However, to realize this approach, several different I/O formats are enforced.

1. In the 1-D transforms along the first dimension, the elements of the initial image vector are pipelined into the array processor and the elements of the intermediate transform vector should be kept inside the processing elements (PEs) of the array processor. This type of array processor is called type I array.
2. In the 1-D transforms along the last dimension, the intermediate data are in the PEs of the array processor and the final results should be pipelined out of the array processor. This type of array processor is called type II array.
3. In the 1-D transforms along all other dimensions, both the intermediate data and the transformation results should stay in the PEs of the array processor. This type of array processor is called type III array.

In order to construct linear array processors for computing the 1-D image transform with different I/O formats, we first derive a 2-D localized data-dependency graph (LDG) of the 1-D image transform (see Fig. 1). Note that the

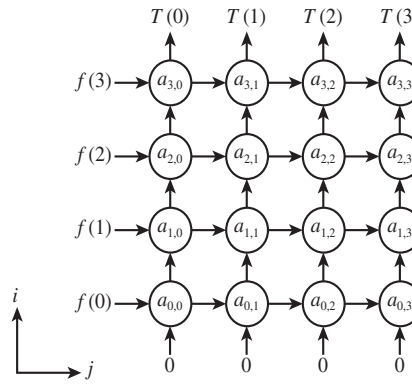


Fig. 1. LDG of 1-D image transform ($N = 4$)

1-D image transform is just a vector-matrix multiplication $\mathbf{T} = \mathbf{fA}$, where $\mathbf{A}$ is an $N \times N$ transformation matrix with $a_{ij} = g(i, j),\ 0 \leq i, j \leq N - 1$, $\mathbf{f}$ is the image vector, and $\mathbf{T}$ is the transformation vector.

If we project the LDG along $\boldsymbol{i}$ direction, we get a type I array (see Fig. 2). Similarly, if we project the LDG along $\boldsymbol{j}$ direction, we get a type II array (see Fig. 3).

A type III array can be constructed by modifying the type I array as follows. Assume that $f(x)$ is in the xth PE initially. Each PE of the type III array has three input and two output channels. The two additional I/O channels are for shifting data $f(x),\ 0 \leq x \leq N - 1$. Datum $f(x)$ is left-shifted to the leftmost PE in x steps, and then, is used for computing $T(u)$ and then right-shifted to the next PE repeatedly at each time step as in type I array. The type III array is shown in Fig. 4.

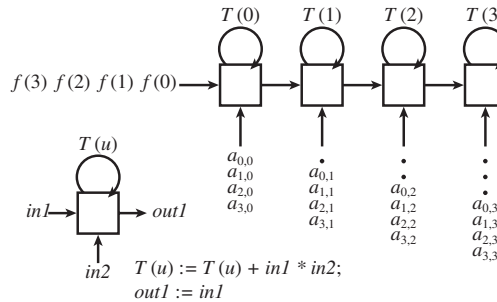


Fig. 2. Type I array for 1-D image transform ($N = 4$)

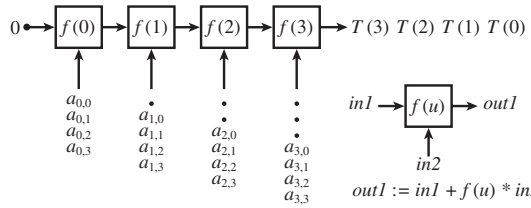


Fig. 3. Type II array for 1-D image transform ($N = 4$)

The computing time is $2N - 1$ for the first two array processors. The computing time for the third array processor is $2N$ since one additional time step is needed to get the initial data $f(x),\ 0 \leq x \leq N - 1$, from the registers inside the PEs of the array.

3 Array Processor Designs for 1-D DCT

In this section, we will refine the three types of linear array processors in the previous section for the 1-D DCT. The main task is to reduce the I/O overhead

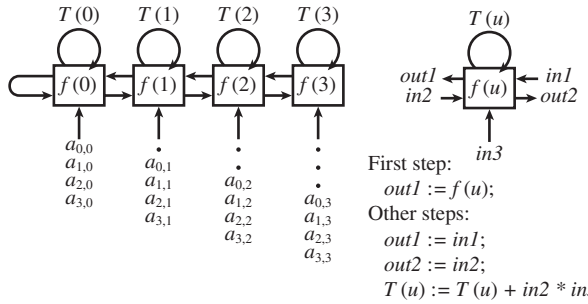


Fig. 4. Type III array for 1-D image transform ($N = 4$)

using the special structure of the DCT kernel. The cost for precomputing the transformation matrix outside the array is very high, and the precomputing also incurs high I/O overhead. Therefore, designs which generate the matrix entries locally while needed are of practical importance.

The 1-D DCT uses the following kernel:

$$g(x,u) = C(u)\sqrt{\frac{2}{N}}\cos(x+\frac{1}{2})\frac{u\pi}{N},$$

where $C(u) = 1/\sqrt{2}$ if $u = 0$, otherwise 1. The 2-D and 3-D DCT kernels are shown below.

$$g(x,y,u,v) = C(u)C(v)\frac{2}{N}\cos(x+\frac{1}{2})\frac{u\pi}{N}\cos(y+\frac{1}{2})\frac{v\pi}{N},$$

and

$$g(x,y,z,u,v,w) = \\ C(u)C(v)C(w)(\tfrac{2}{N})^{3/2}\cos(x+\tfrac{1}{2})\tfrac{u\pi}{N}\cos(y+\tfrac{1}{2})\tfrac{v\pi}{N}\cos(z+\tfrac{1}{2})\tfrac{w\pi}{N}.$$

For video applications, the size of the kernel matrix can be extremely large. How to generate the needed entries of the kernel matrix in the local buffers when perform local DCT computations in each of the active PEs is the key for the design of efficient linear array processor. For simplicity, we omit the scale-factor, $\sqrt{\frac{2}{N}}$, in our designs. The scaling can be done easily at the end of processing.

First, we notice that when we perform $f(x) \times \cos(x+\frac{1}{2})\frac{u\pi}{N}$ in PE(u), the values $\cos(x+\frac{1}{2})\frac{u'\pi}{N}$, $u' < u$, were created and used in the PE(u') already. Some of these values can be delivered through pipelining to PE(u) in order to generate $\cos(x+\frac{1}{2})\frac{u\pi}{N}$. The following formula can be used for this purpose

$$2\cos u\delta\cos\delta = \cos(u+1)\delta + \cos(u-1)\delta.$$

This formula shows that the value $\cos(u+1)\delta$ can be obtained from three values: $\cos u\delta$, $\cos(u-1)\delta$, and $\cos\delta$. Based on this idea, we develop linear array processors which, instead of creating the whole kernel matrix externally, need only an N-vector, $\langle 1, \cos\theta, \cos 3\theta, \ldots, \cos(2N-1)\theta\rangle$, as input. The required entries of the matrix will be calculated locally on the fly inside each PE. How this vector is delivered depends on the types of the linear array processors proposed in the previous section. For the type I array, this vector will be pipelined into the array from left to right as the input vector $\langle f(0), f(1), \ldots, f(N-1)\rangle$. Besides, there are two more channels for the transmissions of intermediate values, $\cos u\delta$ and $\cos(u-1)\delta$, needed for computing $\cos(u+1)\delta$ at PE($u+1$). Since $\cos u\delta$ and $\cos(u-1)\delta$ had been computed at PE(u) and PE($u-1$), respectively, they can be made available in PE($u+1$) through two additional channels, and the proper transmission scheme for right timing during the pipelining computations. The details of the array structure (and its PE functions) for computing 1-DCT based on type I array is depicted in Fig. 5.

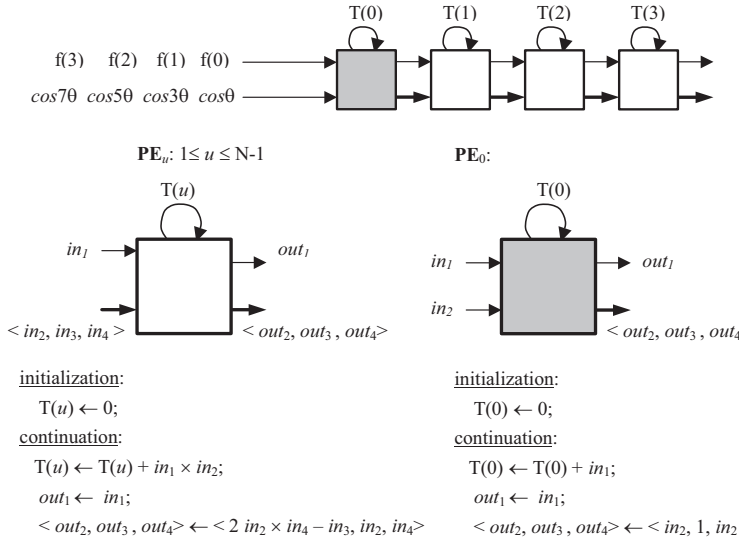


Fig. 5. DCT-I, Type I array for the 1-D DCT ($N = 4$)

The linear array processor of type II for computing 1-D DCT is simple (no additional channels are needed). The two values $\cos u\delta$ and $\cos(u-1)\delta$ are kept now inside PE(u), and are updated and used locally in the next time step. The array structure and the PE functions of the type II array for computing 1-D DCT (DCT-II) are shown in Fig. 6. In each PE of DCT-II, three registers c_0, c_1, and c_2, are used to hold values $\cos\delta$, $\cos(u-1)\delta$, and $\cos u\delta$. An additional register c_3 is used while updating the values of c_1 and c_2.

Finally, the array structure and the PE functions of the type III array for computing 1-D DCT, called DCT-III, are shown in Fig. 7. It follows the same design as in DCT-I.

4 Array Processor for Multidimensional DCT

The 2-D array processor for the 2-D DCT can be constructed using DCT-I and DCT-II on $\boldsymbol{i}$ and $\boldsymbol{j}$ dimensions, respectively. An example array processor, where $N = 4$, is shown in Fig. 8.

This array processor contains $N \times N$ PEs. Each PE can be divided into two parts: functions in left (right) part is the same as the PEs in DCT-I (DCT-II). The input data, $[f(x,y)]_{N\times N}$, are fed into the array column-by-column, and the transformation data, $[T(u,v)]_{N\times N}$, are generated row-by-row. This design allows the intermediate results $z(u,y)$, the results of the 1-D DCTs along the $\boldsymbol{i}$ dimension, to be kept inside the PEs of the array processor. Matrix transposition is not needed. The size of the array is N^2, and the total computing time is $2(N-1) + 2(N-1) = 4N - 2$ time steps.

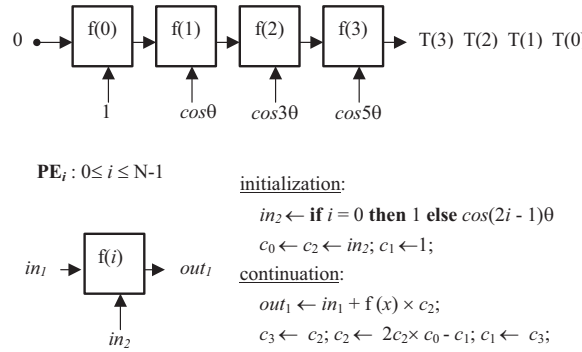


Fig. 6. DCT-II, Type II array for the 1-D DCT ($N = 4$)

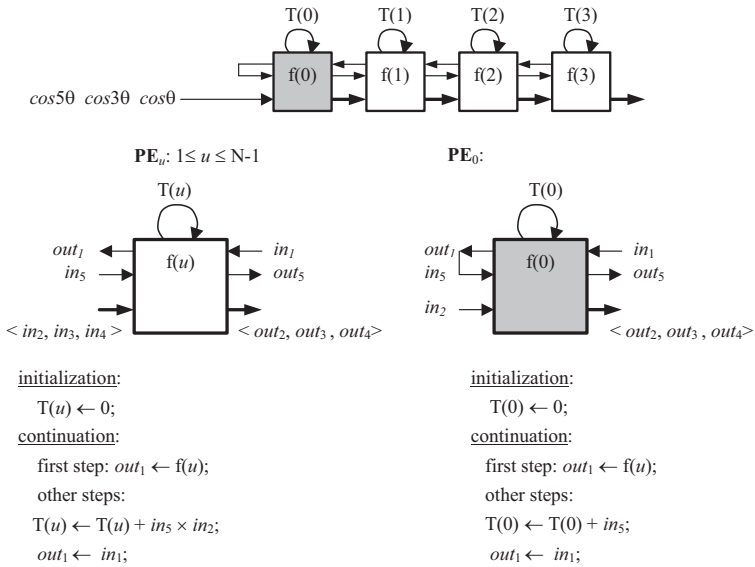


Fig. 7. DCT-III, Type III array for the 1-D DCT ($N = 4$)

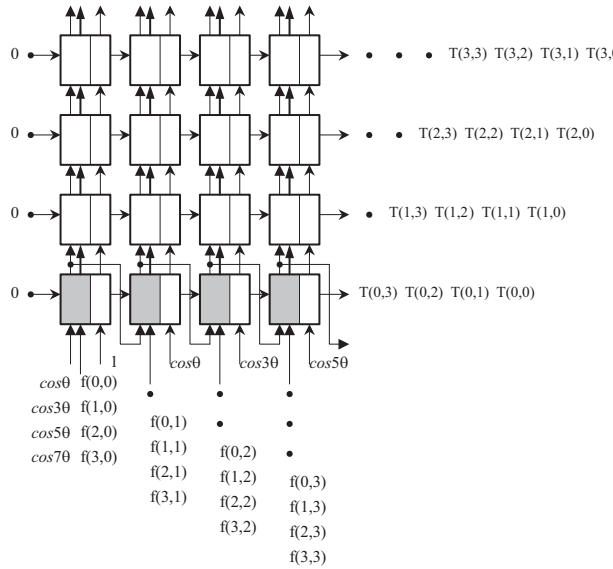


Fig. 8. An example array processor for the 2-D DCT

The 3-D array processor for the 3-D DCT can be constructed similarly using DCT-I, DCT-III, and DCT-II on $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$ dimensions, respectively. Similar to the 2-D case, the PEs of the 3-D array processor are divided into three parts. The functions of each part is equal to that of corresponding PEs in DCT-I, DCT-II, and DCT-III. All proposed array processors can be scaled to different size of problems. Notice that further refinement of the proposed array processors is possible. In this paper, the emphasis is on the key issues of design.

5 Concluding Remarks

In this paper, we proposed a new method for designing array processors for computing multi-dimensional image transforms. We applied this method for the multi-dimensional DCTs which are used widely in video applications. The proposed array processors are highly-regular, time-efficient, fully-pipelined, scalable, and have a small I/O overhead. The advance in VLSI technology will make it possible to develop the 2-D and 3-D array processors proposed in this paper.

References

1. Gonzalez, R.C. and Woods, R.E., *Digital Image Processing*, Addison-Wesley, 1992.
2. Miyazaki, T., Nishitani, T., Edahiro, M., Ono, I., and Mitsuhashi, K., "DCT/IDCT Processor for HDTV developed with DSP Silicon Compiler", *Journal of VLSI Signal Processing*, No. 5, pp. 151–158, 1993.

3. Pratt, W.K., *Digital Image Processing*, John Wiley & Sons, 1991.
4. Rao, K.R. and Yip, P., *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, 1990.
5. Slawecki, D. and Li, W., "DCT/IDCT Processor Design for High-data Rate image Coding", *IEEE Trans. Circuits Syst. Video Techn.*, Vol. 2, pp. 135–146, 1992
6. Sun, M.-T., Chen, T.C., and Gottlieb, A.M., "VLSI Implementation of a 16x16 Discrete Cosine Transform", *IEEE Trans. Circuits Syst.*, Vol. 36, pp. 610–617, 1989.
7. Tonge, G., "Image Processing for Higher Definition Television", *IEEE Trans. Circuits Syst.*, Vol. 34, pp. 1385–1398, 1987.

Design of a Parallel Accelerator for Volume Rendering

Bertil Schmidt

School of Applied Science, Nanyang Technological University, Singapore 639798,
asschmidt@ntu.edu.sg

Abstract. We present the design of a flexible massively parallel accelerator architecture with simple processing elements (PEs) for volume rendering. The underlying parallel computer model is a combination of the SIMD mesh with the instruction systolic array (ISA), an architectural concept suited for easy implementation in very high integration technology. This allows the parallel accelerator unit to be built as a programmable low cost co-processor, that suffices to render volumes with up to 16 million voxels (256^3) at 30 frames per second (fps).

1 Introduction

Volume visualisation [4] is a key technology for the interpretation of 3D scalar data generated by acquisition devices such as biomedical scanners, by supercomputer simulation, or by voxelising geometric models. Especially important for the exploration and understanding of the data are sub-second display rates and instantaneous visual feedback during change of rendering parameters. This is a challenging task due to its rigorous requirements. Firstly, the datasets are very large, typically over 16 MBytes and sometimes exceeding 150 MBytes. Secondly, to be useful the system must be able to produce images at interactive frame rates, preferably at 30 fps, but at least greater than 10 fps.

These tremendous storage and processing requirements have limited the widespread use of volume visualisation. Consequently, research has been conducted towards the development of dedicated volume rendering architectures [11,12,14]. VolumePro [12] is the first single-chip real-time accelerator for consumer PCs. However, the disadvantage of these special-purpose systems is the lack of flexibility with respect to the implementation of different algorithms, e.g. interactive segmentation, feature extraction and other tasks which are to be performed on volume datasets before rendering cannot make use of the special-purpose architecture. ISAs combine the speed and simplicity of systolic arrays with flexible programmability [6, 8], i.e. they achieve extremely high performance cost ratio and can at the same time be used for a wide range of applications, e.g. scientific computing, image processing, multimedia video compression, computer tomography, and cryptography [15-18]. Thus, the ISA architecture fits well for performing high-speed visualisation and processing of 3D datasets at low cost. In this paper we present an ISA architecture that can solve all components of a volume rendering application efficiently by taking advantage of their high degree of inherent parallelism. It has been designed in order to render volumes with up to 256^3 voxels at real-time.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1095-1104, 2000.
© Springer-Verlag Berlin Heidelberg 2000

This paper is organised as follows. Section 2 gives an introduction to volume rendering algorithms. In Section 3 previous SIMD implementations of volume rendering are described. The concept of the ISA is explained in Section 4. Section 5 presents the new accelerator architecture. The parallel algorithms for volume rendering are explained in Section 6 and their performance is evaluated in Section 7. The outlook to further research topics concludes the paper in Section 8.

2 Volume Rendering Algorithms

Volume rendering involves the direct projection of an entire 3D dataset onto a 2D image plane. The data is sampled on a rectilinear grid, represented as a 3D array of volume elements, or *voxels*. Volume visualisation algorithms can simultaneously reveal multiple surfaces, amorphous structures, and other internal structures. These algorithms can be divided into two categories: forward-projection and backward projection. Forward projection algorithms iterate over the dataset during the rendering process projecting voxels onto the image plane. A common forward-projection algorithm is *splatting* [21]. Backward-projection iterates over the image plane during the rendering process by resampling the dataset at evenly spaced intervals along each viewing ray. *Ray casting* [10] is a common backward-projection algorithm.

In ray casting, rays are cast into the dataset. Each ray originates at the viewing position (eye), penetrates a pixel in the image plane (screen), and passes through the dataset. At evenly spaced intervals along the ray, samples are computed using interpolation. The sample values are mapped to display properties such as opacity and colour. A local gradient is combined with a local illumination model at each sample point to provide realistic shading of the object. Final pixel values are found by compositing colour and opacity values along a ray. Composition models the physical reflection and absorption of light. Ray casting offers room for algorithmic improvements by still allowing for high image quality. Several variants of traditional ray casting have been introduced, e.g. [7,23]. The modifications to the original ray casting algorithm to make it more suitable for our parallel accelerator architecture are presented in Section 6.

3 Previous SIMD Volume Rendering Work

Schrder and Stoll proposed an algorithm for the Connection Machine CM2 where the volume is stored one beam per PE. However, the inherent latency of CM2 limited their performance to 4 fps for a 128^3 volume [19]. Yoo et al. presented a method to perform volume rendering on the Pixel Planes 5 machine partly utilising the 2D SIMD mesh and partly the MIMD Graphic processors [24]. They achieved 20 fps for a 128x128x56 volume. Hsu designed a segmented ray casting approach for the DECmpp SIMD mesh [3]. However it distributed the volume in subblocks and only achieved 4-5 fps. Both Vezina [21] and Wittenbrink [23] proposed algorithms for the MASPAR MP-1 (a SIMD 8-connected mesh). Yet, neither achieved frame rates better than 2-5 fps. All of those methods suffered because of the latency inherent in large general-purpose machines. Dogett [2] presented a special-purpose architecture with a 2D array of PEs for volume rendering. However, his PEs are not programmable

ASICs. The PAVLOV design presented in [5] achieves 30 fps for a 256^3 volume on a 64x64 torus of 8-bit-parallel PEs. This architecture is close to our approach since it is a 2D mesh with simple PEs. However, its communication mechanism assumes enough memory to store two times the complete volume on-chip. This is extremely costly in terms of area requirements for PEs as compared to our design.

4 Principle of the ISA

The ISA is a quadratic array of identical processors, each connected to its four direct neighbours by data wires. The array is synchronised by a global clock. The processors are controlled by instructions, row selectors, and column selectors. The instructions are input in the upper left corner of the processor array, and from there they move step by step in horizontal and vertical direction through the array. This guarantees that within each diagonal of the array the same instruction is active during each clock cycle. In clock cycle k+1 processor (i+1,j) and (i,j+1) execute the instruction that has been executed by processor (i,j) in clock cycle k. The selectors also move systolically through the array: the row selectors horizontally from left to right, column selectors vertically from top to bottom. Selectors mask the execution of the instructions within the processors, i.e. an instruction is executed if and only if both selector bits, currently in that processor, are equal to one. Otherwise, a no-operation is executed. This construct leads to a flexible structure, which creates the possibility of very efficient solutions for a large variety of applications, e.g., numeric, image processing, video compression, and cryptography [16-19].

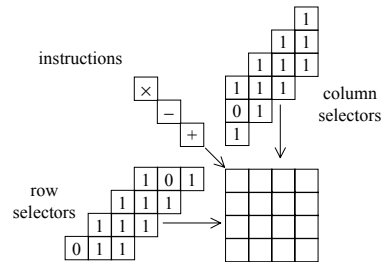


Fig. 1: Control flow in an ISA

Every processor has read and write access to its own memory. Besides that, it has a designated *communication register* (*C-register*) that can also be read by the four neighbour processors. Within each clock phase reading access is always performed before writing access. Thus, two adjacent processors can exchange data within a single clock cycle in which both processors overwrite the contents of their own C-register with the contents of the C-register of its neighbour. This convention avoids read/write conflicts and also creates the possibility to broadcast information across a whole row or column with one single instruction. This property can be exploited for an efficient calculation of row broadcasts, row ringshifts, and row sums, which are the key-operations in many algorithms.

5 Accelerator Architecture Design

Since our aim is to develop a special-purpose architecture for multimedia applications, it is highly desirable to design hardware that can be installed on PCs within the price range of a PC, i.e. to design add-on-boards for PCs. Due to experience gathered in the cause of designing and fabricating the Systola 1024 (also an add-on-board for PCs, with a 32x32 ISA built out of 16 processor chips of 64 processors each [9]), we are in the position of being able to make reliable performance predictions by extrapolation, based on the change of technology parameters. While the Systola 1024 is based on 1.0-micron technology, we are now able to use .25-micron technology, so that on the same chip area that contains 64 processor in case of Systola 1024 we can now place 1024 processors and on a single PC-board we can place 16K processors (together with memory chips, memory multiplexers and a controller with program memory).

As on-chip communication can be clocked at significantly higher frequency than chip-to-chip communication, we have decided to use the 16 processor chips relatively independently, i.e. we assume that the applications allow simple data partitioning. Chip-to-chip communication is done exclusively via locally shared memory, i.e. each processor chip is connected to a memory chip via a simple multiplexer that also allows access to the memories of the four direct neighbours (NEWS) -- here we assume a torus architecture in order to be able to perform easily horizontal and vertical ringshift of data (see Fig. 2). Thus, by avoiding direct off-chip communication we can assume an on-chip clock cycle time of 200 MHz.

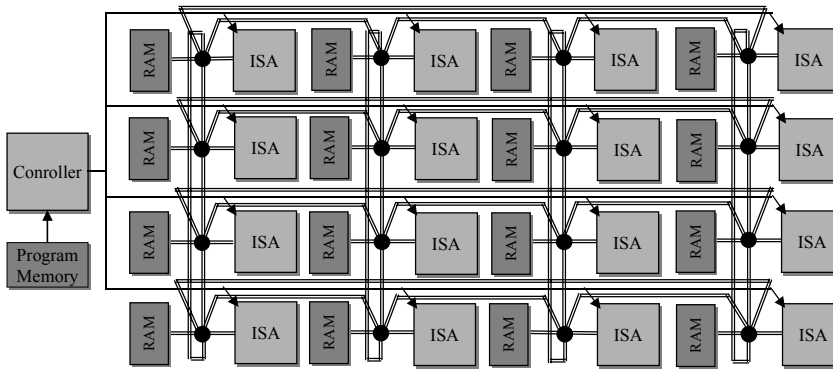


Fig. 2: Data paths of the accelerator architecture

The analysis of ray casting and its processing efforts leads to a fixpoint PE architecture (see Section 6). The PE needs a small local memory for the storage and fast supply of local voxel data. For 8-bit input voxels an intermediate operand length of 16 bits in most computations provides enough accuracy for ray casting [1]. Thus, the wordlength of the data items is set to 16 bits. To allow flexible use of the architecture the PEs must also be able to process longer operands and shorter operands efficiently, e.g. adding two 32-bit numbers in two instructions or adding two

8-bit numbers in one instruction. This idea is incorporated in the design of our computational units. Figure 3 depicts the PE architecture for volume rendering.

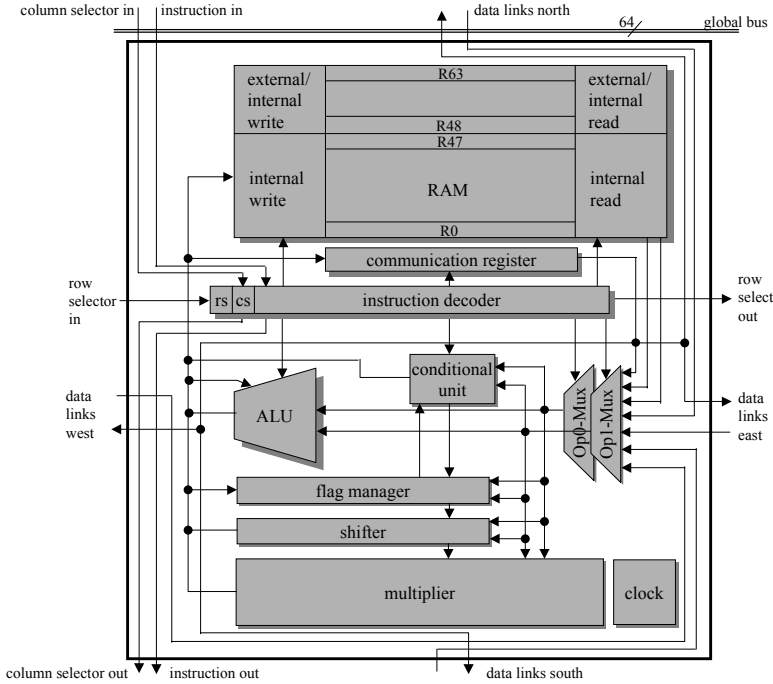


Fig. 3: Block diagram of the processor architecture

Due to the limited chip area the processor has to be very compact. This leads to our choice of a bit-serial data organisation. The bit-serial design allows a higher number of PEs per chip and a higher clock frequency than a corresponding bit-parallel design. The main components of the PE are a set of 64 data registers, a C-register, an ALU, a conditional unit, a multiplier, and a shifter. In addition to the registers there are flags (zero flag, negative flag, activation flag) that control the processing units depending on the state of the processor and several special registers. The wordlength of data items is 16 bits. Because the data is processed bit-serially, the execution of each instruction takes exactly 16 clock cycles. After receiving an instruction, the PE stores it in the instruction register, decodes the two operand addresses and the destination address, retrieves the operands from the register file, executes the instruction, writes the result back to the destination register, and passes the instruction to the next processor. The corresponding instruction set consists of 44 instructions. Since all this is done bit-serially, it can be pipelined on bit-level, such that a new instruction can be fetched and processed every 16 clock cycles.

Extrapolating the design parameters used for Systola 1024 allows us to predict that a 32x32 array of these PEs on a 1cm^2 chip is realistic for a .25-micron CMOS process with a 200 MHz true single phase clock. For a word format of 16 bits the theoretical peak performance for one chip is 12.8 GIPS and for the complete board 204.8 GIPS.

There is already a SIMD single chip architecture with a 32x32 array of bit-serial PEs in .25-micron technology on the market [20]. But the architecture proposed in this paper achieves twice the clock frequency due to adhering as closely as possible to local communication, and its main advantage it gets through its unique control structure that allows the execution of aggregate/reduction functions in a fraction of time as compared to conventional SIMD architectures.

For the fast exchange of data with the processor array each PE has two memory banks. Each memory bank contains 8 interface registers. One of these banks is always assigned to the corresponding processor, the other to a neighbouring memory chip by means of a fast data channel. The exchange of data between ISA and the memory chip is done by bank switching. Both memory banks can be active at the same time, i.e. data transfer can be done concurrently to the execution of an ISA program.

6 Mapping of Ray Casting to the Accelerator Architecture

Fig. 4 shows three possible approaches to parallelising ray casting. According to the form of parallelism that is exploited, we call them *ray*, *beam*, and *slice parallel*.

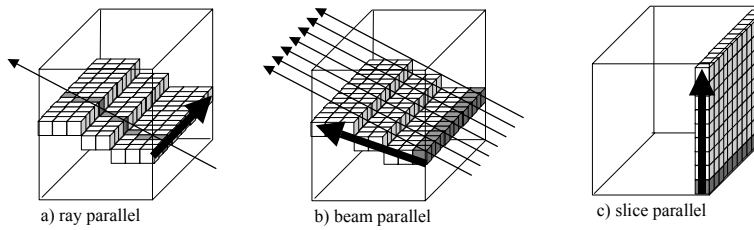


Fig. 4: Three different approaches to parallelising ray casting. Shaded voxels are processed simultaneously. The thick arrows indicate the direction in which the algorithm proceeds.

In the ray parallel approach, all voxels along a ray are processed simultaneously (the shaded voxels in Fig. 4a). The algorithm proceeds ray by ray in scanline order (the thick arrow in Fig. 4a). However, simultaneous access to all voxels along a ray requires irregular data transfer patterns between volume memory and PEs. An alternative to operating on all samples of a single ray is to simultaneously operate on samples of several neighbouring rays. Depending on how the algorithm proceeds, we call these approaches beam parallel (Fig. 4b) and slice parallel (Fig. 4c). A beam is a line of voxels that is parallel to a principle axis of the dataset. The beam parallel ray casting approach follows a group of rays by fetching consecutive beams in major viewing direction. However, the stepping along slanted planes of rays requires complicated addressing mechanisms. The slice parallel approach processes consecutive data slices that are parallel to the *base plane* of the volume dataset (Fig. 4c) and achieves a uniform data access. The base plane is the face of the volume that is closest to perpendicular to the major component of the viewing direction.

A 2D array of ISA PEs can inherently process slice order algorithms very efficiently, since an entire slice of the volume can be processed in parallel. Therefore,

we choose the slice order approach to be mapped on our architecture. Our implementation combines the slice order ray casting approach [1] with segmented ray casting [3] for parallel projections: The volume is partitioned into subcubes. These subcubes are distributed evenly across the memory modules. Each ISA chip computes the colour and opacity values of the portion of the rays, which lie inside the subblock, and writes them into its adjacent memory module. After all subcubes have been processed the segments are composited using chip-to-chip communication. The algorithm consists of the following steps:

Subcube partitioning: Determined by the memory capabilities of PEs, the size of the non-overlapping subcubes is set to 64^3. Each slice is mapped onto a 32x32 ISA by loading 2x2 voxels into each PE. As the algorithm requires a small local neighbourhood of each voxel, three slices are stored in the processor array at any time and processors at the borderline need some data from neighbouring subcubes.

Gradient estimation: The first computing step is the determination of gradients to approximate surface normals for classification and shading. *x*-, *y*-, and *z*-gradient are computed for a voxels sample value $P_{i,j,k}$ at location (i,j,k) using central differences: $G_x = P_{i+1,j,k} - P_{i-1,j,k}$, $G_y = P_{i,j+1,k} - P_{i,j-1,k,}$, $G_z = P_{i,j,k+1} - P_{i,j,k-1}$. Each PE can compute gradients for its 2x2 voxel samples of the current slice in parallel using neighbouring samples. Because the processor array holds three slices at the same time, samples needed from the ahead and behind slice are stored locally in each PE. Samples needed in the two dimensions within the current slice are either also stored locally or in one of the four neighbouring PEs. Other algorithms that use larger neighbourhoods and produce higher quality gradients at additional computational costs can also be mapped efficiently on our architecture. Afterwards gradient magnitude computation continues locally by taking the sum of the squares of the gradient components and then a Newton-Raphson iteration to compute the square root of this value, resulting in an approximation of the gradient magnitude.

Classification: Classification maps a colour and opacity to sample values. Opacity values range from 0.0 (transparent) to 1.0 (opaque). On special-purpose architectures [11,12,14] classification is typically implemented using look-up tables (LUTs). These LUTs are addressed by sample value and gradient magnitude and they output sample opacity and colou. In our architecture using LUTs is not appropriate, as the local memories of PEs are very small. Thus, we are using few low degree polynomials depending on the sample value (for colour) and the product of sample value and gradient (for opacity).

Shading: The Phong shading algorithm [13] is often used in shading subsystems within volume rendering architectures. It requires gradients, light, and reflection vectors to calculate the shaded colour for each sample location. The shading calculation can be expressed as: $I = A + D\ (L*N) + S\ (R*V)^s$, where N is the (normalised) gradient vector, I is the light vector, R is the reflection vector, V is the viewing vector, A, D, L represent ambient, diffuse, and specular material components, and s is the specular exponent. The shading equation can be computed in each PE locally. To normalise the gradient vector we compute the reciprocal of the gradient magnitude by a Newton-Raphson iteration, followed by three multiplications. Parallel view and light vectors are assumed in order to make the reflection independent of the place. Thus, L and V can be stored as constants within each processor. In this case also the computation of the reflection vector can be avoided by using the halfway vector between L and V instead.

Compositing: Compositing is responsible for summing up colour and opacity contributions from interpolated sample locations along a ray into a final pixel colour for display. The front-to-back formulation for composting is $C_{acc} = (1.0 - A_{acc})\ C_{sample} + C_{acc}$ and $A_{acc} = (1.0-A_{acc})\ A_{sample} + A_{acc}$, where C_{acc} is the accumulated colour, A_{acc} is the accumulated opacity, C_{sample} is the interpolated samples colour, and A_{sample} is the interpolated samples opacity.

Since we are processing in slice order fashion, the data is sampled in each slice at the point the ray would intersect the current slice by using bilinear interpolation. Because all the points needed for bilinear interpolation are contained within the slice of voxels currently being processed, it is simpler than trilinear interpolation performed in traditional ray casting. It is also more accurate than nearest neighbour interpolation. As the algorithm moves through the dataset, the point where the ray intersects the current slices moves off the current PE position. This offset is stored and accumulated. Once the ray moves closer to another PE position, the compositing information is shifted to be stored in the corresponding neighbouring PE. In other words, the compositing information of each ray is stored in the PE closest to the ray intersection with the current slice. For parallel projections the corresponding data movement pattern is regular. Thus, whenever a ray attempts to shift to another PE, all the rays in the entire slice buffer shift together.

The rays are cut into segments by the planes that separate the subcubes. These planes are either parallel to the *x*-*y*-plane, or the *x*-*z*-plane, or the *y*-*z*-plane. We refer to these planes as *x*-*y*-planes, *x*-*z*-planes, and *y*-*z*-planes. Without loss of generality we assume that the main viewing axis is the *z*-axis. Firstly, we composite rays that pierce the *x*-*z*-plane between subblocks and then we composite rays that pierce the *y*-*z*-plane. Due to the fact that *z* is the main axis, this can be done in one step. Afterwards compositing has to happen at the *x*-*y*-planes. This can be done for *k* *x*-*y*-planes in $\log_2 k$ steps using a binary tree approach.

Finally, a 2D warp depending on the viewing vector is computed to produce the image for display. Since this is only a 2D operation, it does not influence overall computing time significantly and can be neglected.

7 Performance Evaluation

We execute ray casting within a 256^3 volume by firstly executing ray casting within 64^3 subblocks (subblock processing) and secondly compositing results of rays that move through neighbouring subblocks (final compositing). We have written a C++ cycle accurate simulation of our architecture. During subblock processing we produce the rays slice by slice. Each slice needs 1385 instructions as shown in Table 1. (Table 1 also shows the number of instructions for each substep.)

Table 1: Instruction count (IC) for the ray casting algorithm of Section 6 of one 64x64 slice of a 64^3 subblock with 8-bit voxels on a 32x32 ISA module. For intermediate operands we mostly use a length of 16 bits.

Task	Gradient	Classification	Shading	Compositing	Sum
IC	369	208	504	304	**1385**

Assuming an instruction cycle of 80 ns and computing 64 slices per subblock and 4 subblocks per ISA module, leads to total execution time of 28.4 ms for subblock processing. The data I/O for these steps (based on 150 MBytes/s throughput between each ISA module and RAM) is totally dominated by above computing time and thus can be ignored (see Section 5). Because the final composition step does not require bilinear interpolation it is dominated by the data transfer time. In the worst case (45° viewing angles) it requires 392 KByte per module. The runtime for a 256^3 volume is shown in Table 2. The processing time for larger volumes scales linear with the volume size.

Table 2: Runtime for the rendering of a 256^3 volume with 8-bit voxels on the introduced accelerator architecture. It includes computing time on the ISA and data transfer time between ISA modules and RAM.

Task	Subblock	Final Compositing	Sum
Runtime Accelerator	28.4 ms	2.9 ms	**31.3 ms**

8 Conclusions

In this paper we have presented a massively parallel architecture for volume rendering combining the SIMD computing model with the ISA concept. The accelerator unit has been designed as a co-processor to fit into an inexpensive PC class machine. The global architecture of the accelerator engine has been discussed as well as the detailed implementation of PEs. It has been shown how a volume rendering application can be mapped on the new architecture in order to render a 256^3 volume in real-time.

The introduced architecture is faster, cheaper, and smaller than previous general-purpose SIMD mesh arrays. Different from special-purpose designs, it provides more functionality, e.g. it allows multiple rendering algorithms, and, more importantly, it allows volume processing such as segmentation and feature extraction. The design will give be benefits to a medical or scientific PC where normally users wish to do more than merely render volumetric data. Future work would include identifying applications that profit from this type of processing power. For example, some users may wish to analyse the frequency of local density patterns of a volume and subsequently visualise these measurements. It would be also interesting to study the performance of the new architecture in totally different application areas like scientific computing and multimedia video processing.

References

1. Bitter, I., Kaufman, A.: A Ray-Slice-Sweep Volume Rendering Engine, *Proc. SIGGRAPH/Eurographics'97*, ACM (1997) 121-130
2. Doggett, M.: An array based design for Real-Time Volume Rendering, *Proc. Eurographics'95*, Eurographics (1995) 93-101
3. Hsu, W. M.: Segmented Ray Casting for Data Parallel Volume Rendering, *Parallel Rendering Symposium*, IEEE (1993) 7-14

4. Kaufman, A.: *Volume Visualization*, IEEE CS Press (1991)
5. Kreeger, K., Kaufman, A.: PAVLOV: A Programmable Architecture for Volume Processing, *Proc. SIGGRAPH/Eurographics'98*, ACM (1998) 77-86
6. Kunde, M., et al.: The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, *Parallel Computing* 7 (1988) 25-39
7. Lacroute, P.: Analysis of a Parallel Volume Rendering System Based on the Shear-Warp Factorization, *IEEE Trans. on Visualization and Comp. Graphics* 2 (3) (1996) 218-231
8. Lang, H.-W.: The Instruction Systolic Array, a parallel architecture for VLSI, *Integration, the VLSI Journal* 4 (1986) 65-74
9. Lang, H.-W., Maaß, R., Schimmler, M.: The Instruction Systolic Array - Implementation of a Low-Cost Parallel Architecture as Add-On Board for Personal Computers, *Proc. HPCN 94*, LNCS 797, Springer Verlag (1994) 487-488
10. Levoy, M.: Display of Surfaces from Volume Data, *IEEE Computer Graphics and Applications* 5 (3) (1988) 29-37.
11. Meißner, M., Kanus, U., Straßer, W.: VIZARD II: A PCI-Card for Real-Time Volume Rendering, *Proc. SIGGRAPH/Eurographics'98*, ACM (1998) 61-67
12. Pfister, H., et al.: The Volume Pro Real-Time Ray-Casting System, *Proc. SIGGRAPH'99*, ACM (1999) 251-260
13. Phong: Illumination for Computer Generated Pictures, *Comm. ACM* 18(6) (1975) 311-317
14. Ray, H., et al.: Ray Casting Architectures for Volume Visualization, *IEEE Trans. On Visualization and Computer Graphics*, 5 (3) (1999) 210-223
15. Schimmler, M., Lang, H.-W.: The Instruction Systolic Array in Image Processing Applications, Proc. *Europto 96*, SPIE 2784 (1996) 136-144
16. Schmidt, B., Schimmler, M.: A Parallel Accelerator Architecture for Multimedia Video Compression, *Proc. EuroPar'99*, LNCS 1685, Springer Verlag (1999) 950-959
17. Schmidt, B., Schimmler, M., Schröder, H.: Long Operand Arithmetic on Instruction Systolic Computer Architectures and Its Application to RSA cryptography, *Proc. Euro-Par'98,* LNCS 1470, Springer Verlag (1998) 916-922
18. Schmidt, B., Schimmler, M., Schröder, H.: The Instruction Systolic Array in Tomographic Image Reconstruction Applications, *Proc. PART'98*, Springer Verlag (1998) 343-354
19. Schröder, P., Stoll, G.: Data Parallel Volume Rendering on the MasPar MP-1, *Workshop on Volume Visualization*, ACM (1992) 25-32
20. Teranex Inc.: *Parallel Processing Solves the DTV Format Conversion Problem*, http://www.teranex.com/whitepapers.html (1999)
21. Vezina G., Fletcher, P. A., Robertson, P. K.: Volume Rendering on the MasPar MP-1, *Workshop on Volume Visualization*, ACM (1992) 3-8
22. Westover, L.A.: *Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*, PhD thesis, Dept. of Computer Science, Univ. of South Carolina in Chapel Hill (1991)
23. Wittenbrink, C.M., Somani, A.K.: Time and Space Optimal Data Parallel Volume Rendering using Permutation Warping, *Parallel and Distrib. Comp.* 46(2) (1997) 148-164
24. Yoo, T.S. et al.: Direct Visualization of Volume Data, *IEEE Computer Graphics and Applications* 12 (4) (1992) 63-71

Automated Design of an ASIP for Image Processing Applications

Henjo Schot and Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Technique
P.O. Box 5031, 2600 GA Delft, The Netherlands
H.J.M.Schot@FEL.tno.nl, H.Corporaal@et.tudelft.nl

Abstract. This paper presents the design of highly optimized TTA architectures for image processing applications. An automatic processor design framework as described in [2] is used. Specialized hardware is used to improve the performance-cost ratio of the processors. An explorer searches the design space for solutions that are good in terms of cost and performance. We show that architectures can be found that efficiently execute very different algorithms at low cost. A hardware feasible architecture is presented that efficiently executes a set of image processing algorithms and performs almost equally or better than alternative, commercial-available solutions do.

1 Introduction

In this paper, we show the design of an application specific instruction set processor (ASIP) for a set of image processing algorithms. Processors and code are generated, trying to exploit the instruction level parallelism of image processing algorithms. We show that processors can be generated that efficiently execute very different algorithms at low cost. We add application specific hardware and functionality to improve the performance cost of the processors.

The architecture of the ASIP we develop is a Transport Triggered Architecture (TTA) [2]. An automated design framework, called the MOVE framework [], is used for the development of the VLIW like processor. It tries to find an architecture with an optimal cost/performance ratio. The designer can use Special Function Units (SFUs) to improve the cost-performance ratio of an architecture. These SFUs can be incorporated in the MOVE framework.

Our work on the design of a highly optimized processor architecture differs from others [3] in that we use TTAs and application specific hardware (SFUs) in order to find architectures with higher performance-cost ratios.

The next section describes the image processing algorithms we used. Section 3 shows the mapping of the algorithms to TTAs. Section 4 presents the results and conclusions.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1105-1109, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Image Processing Algorithms

Four image processing algorithms were mapped to TTAs. A color conversion algorithm, and three gray-scale neighborhood algorithms: convolution and two edge detection algorithms on a 3x3 area. Here we concentrate on the color conversion algorithm.

Color conversion is an operation from the area of color image processing. It is used to convert between color representations e.g. a color in RGB has to be converted to represent the same color using CMYK color components. There are several methods to perform color conversion. In our case we uses lookup tables (LUTs) and tri-interpolation.

Using this method, the conversion of a color P starts with searching the lookup table for the eight nearest points. Seven linear interpolations are performed on these points. Figure 1 shows point P and its eight nearest points that correspond with the corners of a cube. The interpolations are shown as bold lines. The final interpolation is performed on V and W. The distance *a* of point V to point P and the distance *1 - a* of point W to point P are used as interpolation coefficients.

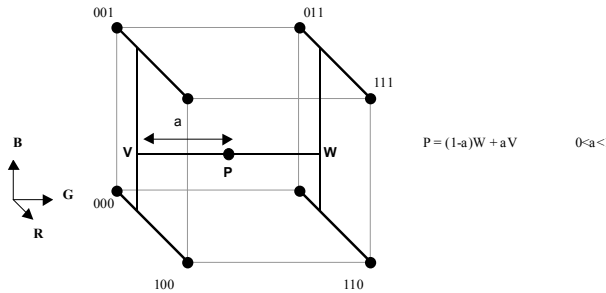


Fig. 1. Tri-linear interpolation of point P in the RGB space.

Since the calculation of each pixel is independent of other pixels, in principle, all pixels can be processed in parallel. The amount of parallelism that can be attained by TTAs is determined by the maximum number of available resources and the ability of the compiler to exploit the parallelism.

In our implementation, we use a LUT of 4 K entries. This size results in reasonable interpolation accuracy at low cost. Higher accuracy requires larger lookup tables. The LUT is addressed using the 4 most significant bits of each color component. The 4 least significant bits are used for the interpolation distance giving interpolation coefficients of 0, 1/16, ... 16/16. We aim to achieve a performance that is comparable or better than that of available solutions (12.5 Mpixels/s [7]), at lower cost.

3 Mapping the Algorithms to TTAs

The main components of the MOVE framework are a retargetable C/C++ compiler, a processor generator and hardware modeller and a design explorer. The explore tool searches the design space for architecture solutions using hardware cost and performance as its main design criteria. The explore tool drives both the compiler and the hardware modeler in order to find architecture solutions with a good performance/cost ratio. A pareto-curve with the resulting architecture solutions is produced, from which the designer chooses an architecture configuration.

The mapping of the algorithm starts with analyzing the solutions that are found in case we use basic operations only. The curve 'w/o SFUs' in figure 2 shows that the latency for high cost architectures remains quite long. E.g.. an architecture of cost 300 (in integer units) does color conversion of a single pixel in 11 cycles.

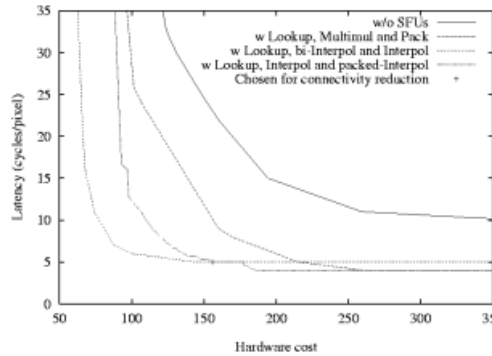


Fig. 2. The TTA design space for the color conversion application.

An enormous improvement in performance is achieved when SFUs are used. Parts of the algorithm that are implemented in hardware are the lookup operation and a linear interpolation. Figure 3 shows the dataflow symbols of different implementations of the interpolation. They are implemented by extending a multiplier FU with these specific functions. The impact of the SFUs on architectures cost/performance is also shown in figure 2. Solutions with 2 to 60 times better performance at equal cost are found.

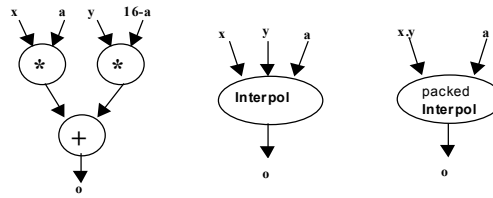


Fig. 3. Dataflow symbols of the interpolation functions

Architectures that execute a set of algorithms are found by combining multiple algorithms in an application. Resulting architectures showed a small loss in

performance for each algorithm, but overall they performed very well, as can be seen in figure 4.

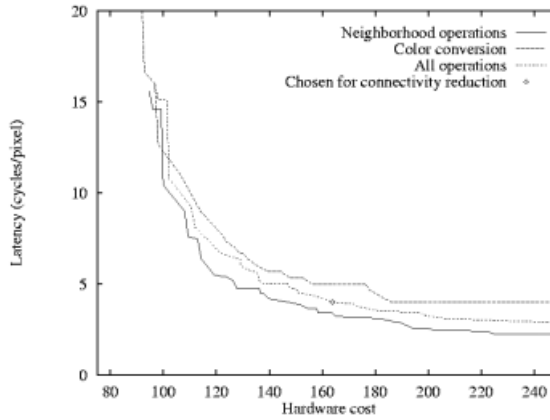


Fig. 4. The TTA design space for both color conversion and neighborhood operations.

4 Results

A feasible processor configuration which efficiently executes the whole set of algorithms is selected from the curve in figure 4. In this configuration, marked with '+', the in- and outputs of each FU and register file are connected to all buses, which is impractical. We therefore remove as much connections as possible without performance losses. The resulting processor is shown in figure5. It can do color conversion at 5.3 cycles/pixel. It contains 8 buses and 11 functional units (FUs). The register file, as shown, contains many read and write ports. However, the tools allow to split up this file into multiple small register files [4][5].

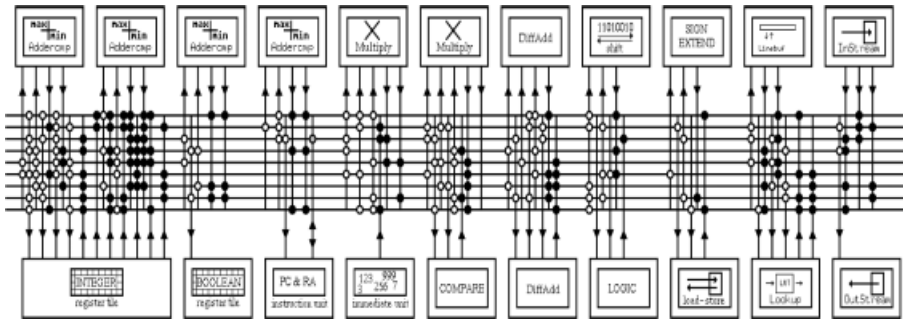


Fig. 5. A processor configuration that efficiently executes the color conversion algorithm and the whole repertoire of neighborhood operations.

Table 1 gives an overview of the performance of commercial available solutions [6]-[11] and our solution. It is seen that the Imagine and the C6x perform better for the convolution application than our solution does. For the other algorithms our solution performs significantly better.

Table 1. Performances of the applications for other available solutions and our solution.

Algorithm	Performance (Mpixels/s)				
	Imagine @ 66 MHz	PixelMagic 44 @ 75 MHz	TI C6201 @ 200 MHz	Chameleon	MOVE @ 100 MHz
convolution	37	22.8	40	n.a.	27
Min-max operation	< 15	11.8		n.a.	27
Edge detection	< 15	< 11.8		n.a.	23
Color conversion	3.0	-	-	12.5	19

5 Conclusion

In this paper, we showed that the MOVE framework can be used to find solutions for digital image processing algorithms. Solutions can be found for applications containing several algorithms, including very different ones. Furthermore, hardware feasible solutions are found that perform almost equally or better than alternative, commercially available solutions.

A large part of the processor design is done automatically. However, a lot of manual interaction is required in the identification and application of Special Function Units. Automation of this part of the design trajectory is currently being researched [1].

References

1. Marnix Arnold and Henk Corporaal, Automatic Detection of Recurring Operation Patterns, Codes '99, May 1999
2. Henk Corporaal, *Microprocessor Architectures; from VLIW to TTA,* John Wiley, 1998, ISBN 0-471-97157-X
3. Joseph A. Fisher, Paolo Faraboschi and Guiseppe Desoli, *Custom-Fit Processors: Letting Applications Define Architectures*,
4. Jan Hoogerbrugge, *Code Generation for Transport Triggered Architectures*, Delft University of Technology, 1996
5. Johan Janssen and Henk Corporaal, *Partitioned Register File for TTAs*, Delft University of Technology
6. Redford, J., Iler, J. and Berger, E., *The PM44: A Single-Chip SIMD GigaOp DSP for Imaging*, Pixel Magic Inc, Andover MA
7. *The Barco Chameleon ASIC; A very high speed, very high accuracy, color correction utility*, Barco Graphics, 1993
8. Redford, J., Iler, J. and Berger, E., *The PM44: A Single-Chip SIMD GigaOp DSP for Imaging*, Pixel Magic Inc, Andover MA
9. *Evaluation of the Performance of the C6201 Processor & Compiler*, Loughborough Sound Images plc., 1996
10. *TMX320C6201 DIGITAL SIGNAL PROCESSOR*, Texas Instruments Inc., 1997
11. *The Imagine engine; Documentation & User Manual*, Arcobel Graphics B.V., March 1994

A Distributed Storage System for a Video-on-Demand Server

Alice Bonhomme and Loïc Prylli

LHPC/ ENS Lyon, 69364 Lyon, France
{Alice.Bonhomme, Loic.Prylli}@ens-lyon.fr

Abstract. The aim of this paper is to present the design of a distributed storage system for a video server. The main goal is to support good fault-tolerance capabilities (no single point of failure, and no perturbation of the clients at the time the failure occurs) while supporting a high-number of video streams.

1 Introduction

The aim of this paper is to present the design of a distributed storage system for a video server. A distributed architecture is quite natural for a video server, given the intrinsic parallelism provided by independent clients on one hand, and the possibility of easily fetching multiple blocks in parallel for a given stream on the other hand. We use a "PC type" cluster, which provides a good performance/price ratio. An important aspect of video servers is the continuity of service even in case of a hardware failure, a distributed architecture provides the required redundancy to handle such failure.

Our goal was to minimize the resource overhead, so a parity strategy is used to manage redundancy rather than mirroring blocks. Then to handle failures without any visible perturbation in the service, the parity block is systematically read just in case (the cost of doing this is low enough). For the intended target of this video-on-demand server, no cache strategy seems possible, so all data to the clients is systematically fetched from disk.

2 Related Works

The subject of video servers (see [GVK+95, GM98, TMDV96]) has been explored in both theory and practice. This is still a active subject because of the diversity of goals. We list below different distributed implementations:

RIO: at UCLA [MSB97] is based on random allocation of striping nodes. This makes it possible to support a wide range of multimedia needs.

MITRA: at USC [GZS+97] focuses on optimizing the throughput by precisely controlling the data placement to optimize the movements of the disk heads.

Server Array: at Eurecom [BB96a] is able to cope with many types of heterogeneity : number of nodes or disks, striping strategy, reliability schemes.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1110–1114, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Tiger: at Microsoft [BB+96b] implements a distributed stream scheduling, replicates data blocks and distributes each copy among a subset of nodes.

From a fault tolerance point of view, these video server implementations exhibits various strategies (parity/mirroring, cf. [GLP98]) and different degrees of fault tolerance (disk, node, network). Tiger uses a distributed replication of the blocks. Mitra chooses to replicate the whole disk and to systematically read on both disks. The Server Array is designed to support any kind of failures using a combination of both strategies. In RIO, the replication scheme of the blocks seems to be more adapted. However, excepted Tiger, all the other prototypes suffer from a single point of failure due to the presence of a meta server, which is in charge of the client connection and the stream scheduling.

3 Overview of the Complete Video Server

The video server (cf. Figure 1) is a cluster of PCs with local storage units. These PCs are interconnected using a Myrinet network. This internal network is independent from the distribution network.

The video server is divided into three parts. The first module manages RTSP client sessions via the distribution network. The second module schedules all real time IO operations and gathers data from the third module: the cluster file system (CFS). This CFS module relies on the internal network (using the GM[1] communication library) to manage the distributed storage (open, close, read and write operations to perform on the cluster nodes). It mainly consists of one IOM (IO Manager) on each node. In fact, all the software components are distributed among all the server nodes.

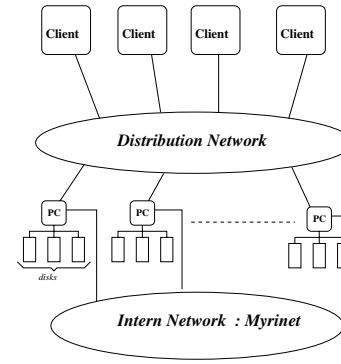


Fig. 1. The video server hardware architecture

4 The Cluster File System

Video data placement and management. We use a striping strategy. A video file is divided into blocks of equal size, distributed among the nodes of the cluster or among a subset of the cluster nodes. Furthermore, for fault tolerance issues, we have parity blocks stored on a node of the subset. When a file is stripped among n nodes, this requires $1/n$ additional space to store the parity blocks. The distribution information for each file is stored in a global table. This table is persistent, and is replicated on each node. To keep it consistent across the nodes, it is accessed through a mutual exclusion scheme. This table also permits

[1] GM: message passing system from Myricom, Inc (`http://www.myri.com/GM`)

to deal with permission problems using a counter for read sessions and write sessions. A local persistent table on each node gives the information about the location of data stored on the local disks.

Implementation of distributed IO. The read function posts an asynchronous read on the distributed file. This function returns an operation descriptor that allows the user to check for the completion of the operation. A read operation is posted from a client to its local IOM, which distributes it among the involved nodes. If necessary, the IOM posts a local read using the local IO subsystem. Remote requests are handled through the internal network. Besides the file's data blocks, the parity block is also systematically requested. If a failure occurs, the parity is already available and no additional delay is needed to get the parity information from the parity node. A status function allows to check for completion of the operation. If all the data has been retrieved, the function returns with a completed status. If one response is missing, the missing data is reconstructed using the parity information and the function returns with the completed status. If more than one response is missing, the function returns with a "not ready" status. The write operation is essentially similar to the read operation. The difference is mainly the addition of the parity blocks generation before sending the request to the remote nodes.

The IOM scheme. The IOM manages at the same time the client requests, the remote IOMs requests and the local accesses. The client requests and the local accesses are stored into two internal queues, the remote IOM requests are stored into a queue fed to the internal network. The IOM scheme consists of polling on each queue and handling the requests. Thus, depending on the request, the IOM sends messages to remote IOMs, calls the local IO subsystem to perform local accesses or updates some local information. The IOM must also keep the context of each client and each operation. Finally it is responsible of the fault tolerance management (cf. Section 5)

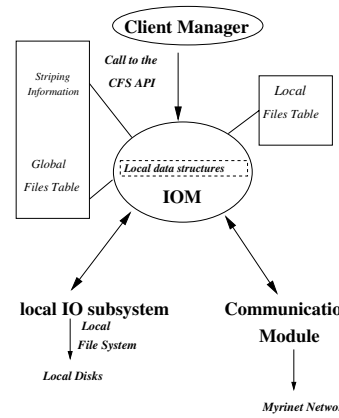


Fig. 2. Interactions between the modules of the CFS

5 Fault Tolerance Management

Within the distributed storage system, it is important that the failure of some component (node, disk, network link) does not cause a server crash. Therefore, we designed a strategy to support any failure (no single point of failure).

Distinction between transient problem and permanent failure. In case of a transient problem lasting less than n seconds (node disconnected from the network for example), the corresponding node is considered as potentially failed, the missing data are computed thanks to parity blocks. Then if the problem disappears, the node comes back in the server without any special treatment. In case a fault is lasting more than n seconds, we rely on the unreachable notification mechanisms of GM to avoid false detection, and the other nodes can consider the unreachable node as permanently failed (each node regularly sends probe messages in order to detect failures even if there are no other useful communications required between the nodes).

Mutual exclusion algorithm. We use a classical algorithm based on the logical clocks defined by Lamport. In order to support a node failure, we modify this algorithm by checking the state of a node that is not responding to mutual exclusion requests or that has failed while owning an exclusive access.

Node recovery. When a node fails, it is important to repair this node without having to stop the server. The recovering node proceeds in several steps: it gets the current global files table and striping table after gaining exclusive access to them, then it can use the striping table and can compare it to its local table to detect video data that needs to be reconstructed, finally it sends read requests to the other nodes to reconstruct the missing data using the parity blocks.

6 Experimental Results

We validated the storage subsystem on a 3-nodes cluster made up of Intel biprocessors connected with a Myrinet network. Each node has four UltraWide/7200 RPM disks in a RAID 0 configuration, accessed using the NTFS file system. What we want to compute is the maximum number of streams for a given bit-rate. We use a test program that uses the CFS to retrieve streams and checks that the real time constraints are met for each block. This allows to experimentally compute the maximum number of streams supported. This experiment is done by executing client requests from one or several nodes, and in presence of a node fault or not (in case of one failure, there can be at most 2 client nodes). The results are shown in Table 3.

	without fault	with fault
1 client node	34	55
2 client nodes	2×32	2×35
3 client nodes	3×23	-

Fig. 3. Number of 4 Mbit/s streams supported on a 3-nodes cluster.

The maximum number of streams globally supported by the server is around 70 which can be shown to match the above hardware using the computation given in [BP00]. The lower part of the array is close to the maximum capacity of the server limited by the local IO capacity. The performance with a fault is better than without, because the redundancy of getting one extra remote block is avoided.

7 Conclusion

The aim of this paper was to describe the overall design of a distributed storage system targeted at video. The choices that have been made are consistent with the intended target: reliably delivering video while supporting as many streams as possible even in case of failure.

If redundancy strategies and fault recovery have been well studied in previous works, in practice the problem of fault detection of a host must be correctly handled: the design should guarantee a consensus between all other hosts before deciding that another one has failed (for the purpose of mutual exclusion), but this consensus can take time. The solution proposed here allows the delivery of ongoing streams without perturbation (handling parity blocks does not depend on mutual exclusion and thus on host fault detection), independently of the time needed to detect reliably an host failure.

References

[BB96a] C. Bernhardt and E. Biersack. *High-Speed Networking for Multimedia Applications*, chapter The Server Array: A Scalable Video Server Architecture. Kluwer, 1996.

[BB+96b] William J. Bolosky, Joseph S. Barrera, et al. The Tiger video fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*. IEEE Computer Society, April 1996.

[BP00] Alice Bonhomme and Loic Prylli. A distributed storage system for a video-on-demand server. Technical Report 2000-16, LIP, ENS Lyon, France, April 2000.

[GLP98] L. Golubchik, J. Lui, and M. Papadopouli. A survey of approaches to fault tolerant design of VOD servers: Techniques, analysis and comparison. *Parallel Computing*, 24(1):123–155, 1998.

[GM98] S. Ghandeharizadeh and R. Muntz. Design and implementation of scalable continuous media servers. *Parallel Computing*, 24:91–122, 1998.

[GVK+95] D.J. Gemmel, H.M. Vin, D.D. Kandlur, P.V. Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, 28(5):40–49, November 1995.

[GZS+97] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and A.W Li. Mitra : A scalable continuous media server. *Multimedia Tools and Applications Journal*, 5(1):79–108, July 1997.

[MSB97] R. Muntz, J.R Santos, and S. Berson. RIO: A real-time multimedia object server. *ACM Performance Evaluation Review*, 25(2):29–35, September 1997.

[TMDV96] R. Tewari, R. Mukherjee, D.M. Dias, and H.M. Vin. Design and performance tradeoffs in clustered video servers. In *the IEEE international Conference on Multimedia Computing and Systems (ICMCS'96)*, pages 144–150, May 1996.

Topic 18
Cluster Computing

Rajkumar Buyya, Mark Baker, Daniel C. Hyde, and Djamshid Tavangarian

Topic Chairmen

Recent Advances in Cluster Computing

Cluster computing can be described as a fusion of the fields of parallel, high-performance, distributed, and high-availability computing. It has become a popular topic of research among the academic and industrial communities, including system designers, network developers, algorithm developers, as well as faculty and graduate researchers. The use of clusters as a application platform is not just limited to the scientific and engineering area; there are many business applications, including E-commerce, that are benefiting from the use of clusters.

There are many exciting areas of development in cluster computing. These include new ideas as well as hybrids of old ones that are being deployed in production and research systems. There are attempts to couple multiple clusters, either located within one organisation or situated across multiple organisations forming what is known as a federated clusters or hyperclusters. The exploitation of federated clusters (clusters of clusters) as an infrastructure can seem to be approaching the area of the increasingly popular GRID infrastructure.

The concept of portals that offers web-based access to applications running on clusters is becoming widely accepted. Such computing portals, offering access to scientific applications online, are known as scientific portals. PAPIA (Parallel Protein Information Analysis system) developed by the Japanese Real World Computing Partnership (RWCP) is an example of one. PAPIA allows scientists to have online access to a Protein Data Bank (PDB) in order to perform protein analysis on clusters. Anyone can perform the analysis of protein molecules and genetic DNA sequence from anywhere, at any time using any platform. In the future, we will see many such applications that exploit clustering and the Internet technologies for scientific discovery.

The TopClusters project is TFCC collaboration with the TOP500 team. Plans are underway to build a database to record the performances of the most powerful cluster systems in different areas. These is great interest in this area as clusters are being used as platforms to host a diverse range applications, including scientific computing, web serving, database, and mission critical systems. Each of these areas has its own specific requirements. For example scientific applications are driven by floating-point performance whereas database applications are driven primarily by system I/O performance. The TopClusters project will measure these parameters (Mflop/s, I/O, TPC, MTBF, etc.) and use this to try and understand more fully the key aspects that need to addressed when building clusters for new and emerging applications.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1115–1117, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The IEEE Task Force on Cluster Computing (TFCC) is acting as a focal point and guide to the current cluster computing community. The TFCC has been actively promoting the field of cluster computing with the aid of a number of novel projects, for example we have an educational activity that has a book donation programme, holds forums for informal discussion, helps guide R&D work both in academic and industrial settings through workshops, symposiums and conferences.

The recent developments in high-speed networking, middleware and resource management technologies have pushed clusters into the mainstream as general purpose computing system. This is clearly evident from the use of clusters as a computing platform for solving problems in number of disciplines. It also raises a number of challenges that cluster systems need to address with respect to their ability to support for example:

- System architecture.
- Heterogeneity.
- Single system image.
- System scalability.
- Resource management.
- System administration.
- Performance.
- Reliability.
- Application scalability.
- Management and administration of hyperclusters.

Based on the aspects allready mentioned, a number of challenges listed above are among the issues addressed by the 12 research papers that we have selected from 27 contributions for the Euro-Par 2000 Cluster Computing Workshop. The program of the Workshop presents articles which demonstrate both theoretical and practical results of research works and new developments regarding cluster computing. The following papers have been accepted for presentation and discussion in the workshop, they cover:

- F. Rauch, C. Kurmann and T. M. Stricker presents an analytical model that guides an implementation towards an optimal configuration for any given PC cluster. The model is validated by measurements on a cluster using Gigabit- and Fast Ethernet links.
- W. Hu, F. Zhang, and H. Liu describe topics belonging to the SMP and DSMP topics. They introduce an SMP protocol for the home-based software DSM system JIAJIA.
- R. Cunniffe and B. A. Coghlan proposes a framework for cluster management which enables a cluster to be more efficiently utilized within a research environment.
- V. Shurbanov, D. Avresky, P. Mehra and W. Watson describe in their paper the performance implications of several end-to-end flow-control schemes clusters based on the ServerNet system-area network.
- The authors of the next paper H. Pedroso and J. G. Silva introduce the system WMPI as the first implementation of the MPI standard for Windows based machines.
- The goal of the paper of F. Solsona, F. Giné, P. Hernández and E. Luque is to build a NOW that runs parallel applications with performance equivalent

to a MPP as well as executing sequential tasks as a dedicated uniprocessor with acceptable performance.
- Z. Juhasz and L. Kesmarki investigates the possible use of Jini technology for building Java-based metacomputing environments and gives an overview of Jini and highlights those features that can be used effectively for metacompting.
- The implementation of a skeleton library allowing the C programmer to write parallel programs using skeleton abstractions to structure and exploit parallelism is given by M. Danelutto and M. Stigliani.
- The contributions of the paper of C. Wagner and F. Mueller are twofold. First, a protocol for distributed mutual exclusion is introduced using a token-based decentralized approach, which allows either multiple concurrent readers or a single writer to enter their critical sections. Second, this protocol is evaluated in comparison with another protocol that uses a static structure instead of dynamic path compression.
- W. Schreiner, C. Mittermaier and F. Winkler describe a parallel solution to the problem of reliably plotting a plane algebraic curve based on Distributed Maple, a distributed programming extension written in Java.
- An Application Programming Interface (PCI-DDC) is described by E. Renault, P. David and P. Feautrier which provides different levels of integration in the kernel depending on the security and the performances expected by the administrator.
- A Clustering Approach for Improving Network Performance in Heterogeneous Systems is the topic of the presentation of V. Arnau, J. M. Orduña, S. Moreno, R. Valero and A. Ruiz. They propose on one hand a clustering algorithm that, given a network topology, provides a network partition adapted to the communication requirements of the set of applications running on the machine. On other hand, they propose a criterion to measure the quality of each one of the possible mappings of processes to processors that the provided network partition may generate.

All in all, we think we have an interesting program for this scientific event on cluster computing.

We are grateful to the authors for their contributions and the reviewers who provided many useful comments in a very short time. We are also grateful to the organizers of the Euro-Par Conference 2000 for their helpful support regarding the organization of the Workshop.

Partition Cast — Modelling and Optimizing the Distribution of Large Data Sets in PC Clusters

Felix Rauch, Christian Kurmann, and Thomas M. Stricker

Laboratory for Computer Systems
ETH - Swiss Institute of Technology
CH-8092 Zürich, Switzerland
{rauch,kurmann,tomstr}@inf.ethz.ch

Abstract. Multicasting large amounts of data efficiently to all nodes of a PC cluster is an important operation. In the form of a *partition cast* it can be used to replicate entire software installations by cloning. Optimizing a partition cast for a given cluster of PCs reveals some interesting architectural tradeoffs, since the fastest solution does not only depend on the network speed and topology, but remains highly sensitive to other resources like the disk speed, the memory system performance and the processing power in the participating nodes. We present an analytical model that guides an implementation towards an optimal configuration for any given PC cluster. The model is validated by measurements on our cluster using Gigabit- and Fast Ethernet links. The resulting simple software tool, *Dolly*, can replicate an entire 2 GByte Windows NT image onto 24 machines in less than 5 minutes.

1 Introduction and Related Work

The work on partition cast was motivated by our work with the Patagonia multi-purpose PC cluster. This cluster can be used for different tasks by booting different system installations [12]. The usage modes comprise traditional scientific computing workloads (Linux), research experiments in distributed data processing (data-mining) or distributed collaborative work (Linux and Windows NT) and computer science education (Windows NT, Oberon). For best flexibility and maintenance, such a multi use cluster must support the installation of new operating system images within minutes.

The problem of copying entire partitions over a fast network leads to some interesting tradeoffs in the overall design of a PC cluster architecture. Our cluster nodes are built from advanced components such as fast microprocessors, disk drives and high speed network interfaces connected via a scalable switching fabric. Yet it is not obvious which arrangement of the network or which configuration of the software results in the fastest system to distribute large blocks of data to all the machines of the cluster.

After in depth analytical modelling of network and cluster nodes, we create a simple, operating system independent tool that distributes raw disk partitions. The tool can be used to clone any operating system. Most operating systems can perform automatic installation and customization at startup and a cloned partition image can therefore be used immediately after a partition cast completes.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1118–1131, 2000.
© Springer-Verlag Berlin Heidelberg 2000

For experimental verification of our approach we use a meta cluster at ETH that unites several PC clusters, connecting their interconnects to a dedicated cluster backbone. This cluster testbed offers a variety of topologies and networking speeds. The networks include some Gigabit networking technology like SCI [7, 5], Myrinet [3] with an emphasis on Fast and Gigabit Ethernet [13]. The evaluation work was performed on the Patagonia sub-cluster of 24 Dell 410 Desktop PCs configured as workstations with keyboards and monitors. The Intel based PC nodes are built around a dual Pentium II processor configuration (running at 400 MHz) and 256 MB SDRAM memory connected to a 100 MHz front side bus. All machines are equipped with 9 GB Ultra2 Cheetah SCSI harddisk drives which can read and write a data stream with more than 20 MByte/s.

Partition cloning is similar to general backup and restore operations. The differences between logical and physical backup are examined in [8]. We wanted our tool to remain operating system and file system independent and therefore we work with raw disk partitions ignoring their filesystems and their content.

Another previous study of software distribution [9] presents a protocol and a tool to distribute data to a large number of machines while putting a minimal load on the network (i.e. executing in the background). The described tool uses unicast, multicast and broadcast protocols depending on the capabilities and the location of the receivers. The different protocols drastically reduce the network usage of the tool, but also prevent the multicast from reaching near maximal speeds.

Pushing the protocols for reliable multicast over unreliable physical network towards higher speeds leads to a great variation in the perceived bandwidth, even with moderate packet loss rates, as shown in [11]. Known solutions for reliable multicast (such as [6]) require flow-control and retransmission protocols to be implemented in the application. Most of the multicast protocol work is geared to distribute audio and video streams with low delay and jitter rather than to optimize bulk data transfers at a high burst rate.

The model for partition cast is based on similar ideas presented in the throughput-oriented copy-transfer model for MPP computers [14].

A few commercial products are available for operating system installation by cloning, such as *Norton Ghost* [1], *ImageCast* [2] or *DriveImagePro* [3]. All these tools are capable of replicating a whole disk or individual partitions and generating compressed image files, but none of them can adapt to different networks or the different performance characteristics of the computers in PC clusters. Commercial tools also depend on the operating- and the file system, since they use knowledge of the installed operating- and file systems to provide additional services such as resizing partitions, installing individual software packages and performing customizations.

An operating system independent open source approach is desired to support partition cast for maintenance in Beowulf installations [2]. Other applications of our tool could include *presentation-, database-* or *screen image cast* for new applications in distributed data mining, collaborative work or remote tutoring on clusters of PCs. An early

[1] Norton Ghost©, Symantec, *http://www.symantec.com/*

[2] ImageCast©, Innovative Software Ltd., *http://www.innovativesoftware.com/*

[3] DriveImagePro©, PowerQuest, *http://www.powerquest.com/*

survey about research in that area including video-cast for clusters of PCs was done in the Tiger project [4].

2 A Model for Partition-Cast in Clusters

In this Chapter we present a modelling scheme that allows to find the most efficient logical topology to distribute data streams.

2.1 Node Types

We divide the nodes of a system into two categories, active nodes which duplicate a data stream and passive nodes which can only route data streams. The two node types are shown in Figure 1.

Active Node A node which is able to duplicate a data stream is called an active node. Active nodes that participate in the partition cast store the received data stream on the local disk.

An active node has at least an in-degree of 1 and is capable of passing the data stream further to one or more nodes (out-degree) by acting as a T-pipe.

Passive Node A passive node is a node in the physical network that can neither duplicate nor store a copy of the data stream. Passive nodes can pass one or more streams between active nodes in the network.

Fig. 1. An active node (left) with an in-degree of 1 and an out-degree of 2 as well as a passive node (right) with an in- and out-degree of 3.

Partition cast requires *reliable* data streams with flow control. Gigabit Ethernet switches do only provide *unreliable* multicast facilities and must therefore be modelled as passive switches that do only route TCP/IP point-to-point connections. Incorporating intelligent network switches or genuine broadcast media (like Coax Ethernet or Hubs) could be achieved by making them active nodes and modelling them at the logical level. This is only an option for expensive Gigabit ATM switches that feature multicast capability on logical channels with separate flow control or for simple switches that are enhanced by a special end-to-end multicast protocol that makes multicast data transfers reliable.

2.2 Network Types

The different subsystems involved in a partition-cast must be specialized to transfer long data streams rather than short messages. Partitions are fairly large entities and our model is therefore purely bandwidth-oriented. We start our modelling process by investigating the topology of the physical network and taking a note of all the installed link and switch capacities.

Physical Network The physical network topology is a graph given by the cables, nodes and switches installed. The vertices are labeled by the maximal switching capacity of a node, the edges by the maximal link speeds.

The model itself captures a wide variety of networks including hierarchical topologies with multiple switches. Figure 2 shows the physical topology of the meta cluster installed at ETH and the topology of our simple sub-cluster testbed. The sub-cluster testbed is built with a single central Gigabit Ethernet switch with *full duplex* point-to-point links to all the nodes. The switch has also enough Fast Ethernet ports to accommodate all cluster nodes at the low speed. Clusters of PCs are normally built with simple and fast layer-2 switches like our Cabletron Smart Switch Routers. In our case the backplane capacity for a 24 port switch is at 4 GByte/s and never results in a bottleneck.

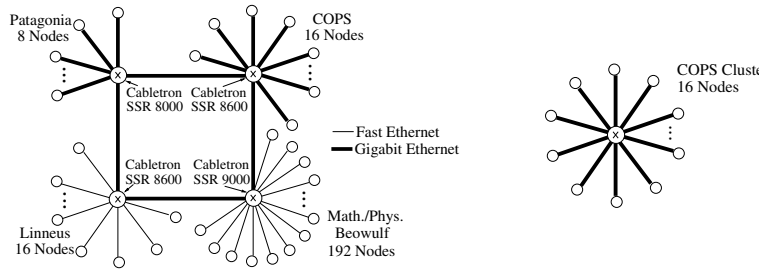


Fig. 2. Physical network topologies of the ETH Meta-Cluster (left) and the simple sub-cluster with one central switch (right).

Our goal is to combine several subsystems of the participating machines in the most efficient way for an optimal partition-cast, so that the cloning of operating system images can be completed as quickly as possible. We therefore define different setups of logical networks.

Logical Network The logical network represents a connection scheme, that is embedded into a physical network. A spanning tree of TCP/IP connections routes the stream of a partition cast to all participating nodes. Unlike the physical network, the logical network must provide reliable transport and flow control over its channels.

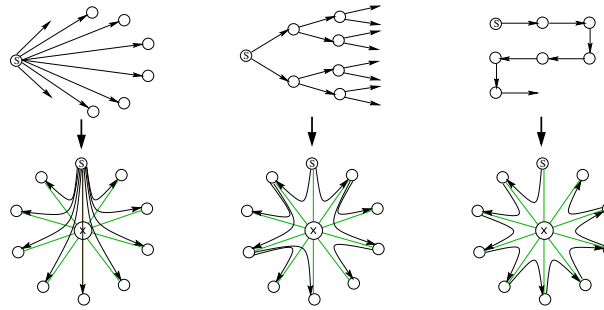


Fig. 3. Logical network topologies (top) describing logical channels (star, n-ary spanning tree, multi-drop-chain) and their embedding in the physical networks.

Star A logical network with one central server, that establishes a separate logical channel to all n other nodes. This logical network suffers heavy congestion on the outgoing link of the server.

n-ary Spanning Tree Eliminates the server-bottleneck by using an *n-ary spanning tree* topology spanning all nodes to be cloned. This approach requires active T-nodes which receive the data, store it to disk and pass it further to up to n next nodes in the tree.

Multi-Drop-Chain A degenerated, specialized tree (unary case) where each active node stores a copy of the stream to disk and passes the data to just one further node. The chain is spanning all nodes to be cloned.

Figure 3 shows the above described topologies as well as their embedding in the physical networks. We assume that the central switch is a passive node and that it cannot duplicate a partition cast stream.

2.3 Capacity Model

Our model for maximal throughput is based on capacity constraints expressed through a number of inequalities. These inequalities exist for active nodes, passive nodes and links, i.e. the edges in the physical net. As the bandwidth will be the limiting factor, all subsystems can be characterized by the maximal bandwidth they achieve in an isolated transfer. The extended model further introduces some more constraints e.g. for the CPU and the memory system bandwidth in a node (see Section 2.5).

Reliable transfer premise We are looking for the fastest possible bandwidth with which we can stream data to a given number of active nodes. Since there is flow control, we know that the bandwidth b of the stream is the same in the whole system.

Fair sharing of links We assume that the flow control protocol eventually leads to a stable system and that the links or the nodes, dealing with the stream, allocate the bandwidth evenly and at a precise fraction of the capacity.

Both assumptions hold in the basic model and will be slightly extended in the refined model that can capture raw and compressed streams at different rates simultaneously.

Edge Capacity defines a maximum streaming capacity for each physical link and logical channel (see Figure 4).

As the physical links normally operate in *full duplex mode*, the inbound- and outbound-channels can be treated separately. If the logical-to-physical mapping suggests more than one logical channel over a single physical link, its capacity is evenly shared between them. Therefore the capacity is split in equal parts by dividing the link capacity through the number of channels that are mapped to the same physical link.

Example: For a binary tree with in-degree 1 and out-degree 2 mapped to one physical Gigabit Ethernet link the bandwidth of a stream has to comply with the following edge inequality:

$$ {}_1E_2: \quad b < 125, \quad 2b < 125 \quad \rightarrow \quad b < \frac{125}{2} \tag{1} $$

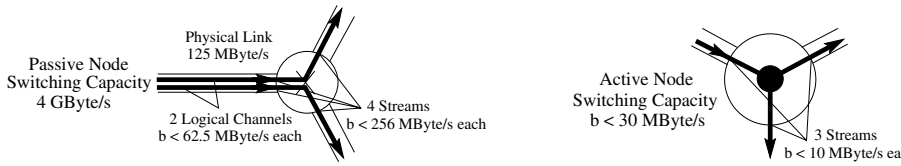


Fig. 4. Edge capacities exist for the physical and logical network, node capacities for each in- and out-stream of a node.

Node Capacity is given by a switching capacity a node can provide, divided through the number of streams it handles.

The switching capacity of a node can be measured experimentally (by parameter fitting) or be derived directly from data of the node computer through a detailed model of critical resources. The experimental approach provides a specific limit value for each type of active node in the network, i.e. the maximal *switching capacity*. Fitting all our measurements resulted in a total switching capacity of 30 MB/s for our active nodes running on a 400 MHz Pentium II based cluster node. The switching capacity of our passive node, the 24 port Gigabit Ethernet switch is about 4 GByte/s - much higher than needed for a partition cast.

2.4 Model Algorithm

With the model described above we are now able to evaluate the different logical network alternatives described earlier in this Section of the paper. The algorithm for evaluation of the model includes the following steps:

```
algorithm basicmodel
  chose the physical network topology
  chose the logical network topology
  determine the mapping and the edge congestions
  for all edges
      determine in-degree, out-degree of nodes attached to edge
      evaluate channel capacity (according to logical net)
  for all nodes
      determine in-degree, out-degree and disk transfer of the node
      evaluate node capacity
  solve system of inequalities and find global minimum
  return minimum as achievable throughput
```

Example: We compare a multi-drop-chain vs. the n-ary spanning tree structure for Gigabit Ethernet as well as for Fast Ethernet. The chain topology with all active nodes with in-degree ι and out-degree ω of exactly one (except for the source and the sink) can be considered as a special case of an unary tree (or hamiltonian path) spanning all the active nodes receiving the partition cast.

- *Topology:* We evaluate the logical n-ary tree topology of Figure 3 with 5 nodes (and a streaming server) mapped on our simple physical network with a central switch of Figure 2. The out-degree shall be variable from 1 to 5, multi-drop-chain to star.
- *Edge Capacity:* The in-degree is always 1. The out-degree over one physical link varies between 1 for the multi-drop-chain and 5 for the star which leads to the following inequality:

$$ {}_1E_o: \quad ob < 125 \quad \rightarrow \quad b < \frac{125}{o} \quad \text{for Gigabit Ethernet} \tag{2} $$

$$ {}_1E_o: \quad ob < 12.5 \quad \rightarrow \quad b < \frac{12.5}{o} \quad \text{for Fast Ethernet} \tag{3} $$

- *Node Capacity N:* For the active node we take the evaluated capacity of 30 MByte/s with the given in-degree and out-degree and a disk write:

$$ N_{1,o,1}: \quad (1+o+1)b < 30 \quad \rightarrow \quad b < \frac{30}{(1+o+1)} \tag{4} $$

We now label all connections of the logical network with the maximal capacities and run the algorithm to find a global minimum of achievable throughput. The evaluation of the global minimum indicates that for Gigabit Ethernet the switching capacity of the active node is the bottleneck for the multi-drop-chain and for the n-ary trees. But for the slower links of a Fast Ethernet the n-ary tree the network rapidly becomes a bottleneck as we move to higher branching factors. Section 4 gives a detailed comparison of modelled and measured values for all cases considered.

2.5 A More Detailed Model for an Active Node

The basic model considered two different resources: Link capacity and switching capacity. The link speeds and the switch capacity of the passive node were taken from the

physical data sheets of the networking equipment, while the total switching capacity of an active node was obtained from measurements by a parameter fit. Link and switching capacity can only lead the optimization towards a graph theoretical discussion and will only be relevant to cases that have extremely low link bandwidth and high processing power or to systems that are trivially limited by disk speed. For clusters of PCs with high speed interconnects this is normally not the case and the situation is much more complex. Moving data through I/O buses and memory systems at full Gigabit/s speed remains a major challenge in cluster computing. The systems are nearly balanced between CPU performance, memory system and communication speed and some interesting tradeoffs can be observed. As indicated before, several options exist to trade off the processing power in the active node against a reduction of the load on the network. Among them are data compression or advanced protocol processing that turns some unreliable broadcast capabilies of Ethernet switches into a reliable multicast.

For a better model of an active node we consider the data streams *within* an active node and evaluate several resource constraints. For a typical client the node activity comprises receiving data from the network and writing partition images to the disk. We assume a "one copy" TCP/IP protocol stack as provided by standard Linux. In addition to the source and sink nodes the tree and multi-drop chain topologies require active nodes that store a data stream and forward one or more copies of the data streams back into the network. Figure 5 gives a schematic data flow in an active node capable of duplicating a data stream.

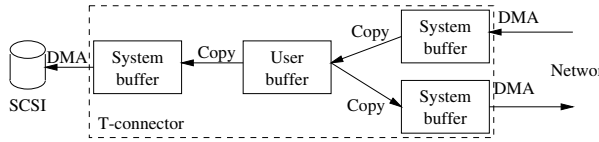


Fig. 5. Schematic data flow of an active node running the *Dolly* client.

2.6 Modelling the Limiting Resources in an Active Node

The switching capacity of an active node is modelled by the two limits of the network and four additional resource limits within the active node.

Link capacity as taken from the physical specifications of the network technology (125 MB/s (Gigabit Ethernet) or 12.5 MB/s (Fast Ethernet) on current systems).

Switch capacity of passive nodes as taken from the physical specifications of the network hardware (2 or 4 GB/s depending on the Cabletron Smart Switch Router model, 8000 or 8600).

Disk system similar to a link capacity in the basic model (24 MB/s for a Seagate Cheetah 10'000 RPM disk).

I/O bus capacity the sum of data streams traversing the I/O bus must be less than its capacity (132 MB/s on current, 32 bit PCI bus based PC cluster nodes).

Memory system capacity the sum of the data streams to and from the memory system must be less than the memory system capacity (180 MB/s on current systems with the Intel 440 BX chipset).

CPU Utilization the processing power required for the data streams at the different stages. For each operation fraction coefficient $1/a_1, 1/a_2, 1/a_3, \ldots$ indicates the maximal speed of the stream with exclusive use of 100% CPU. The sum of the fractions of CPU use must be $< 1 (= 100\%)$ (Fractions considered: 80 MB/s SCSI transfer, 90 MB/s internal copy memory to memory, 60 MB/s send or receive over Gigabit Ethernet, 10 MB/s to decompress a data stream for a current 400 MHz single CPU cluster node).

Limitations on the four latter resources result in constraint inequalities for the maximal throughput achievable through an active node. The modelling algorithm determines and posts all constraining limits in the same manner as described in the example with a single switching capacity. The constraint over the edges of a logical network can be evaluated then into the maximum achievable throughput considering all limiting resources.

2.7 Dealing with Compressed Images

Partition images or multimedia presentations can be stored and distributed in compressed form. This reduces network load but puts an additional burden on the CPUs in the active nodes. Compressing and uncompressing is introduced into the model by an additional data copy to a *gunzip* process, which uncompresses data with an output data rate of about 10 MByte/s (see Figure 6).

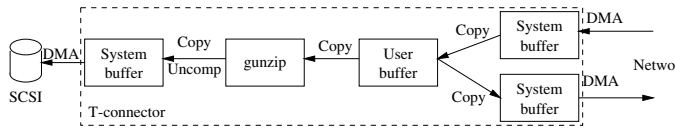


Fig. 6. Schematic data flow of a *Dolly* client with data decompression.

The workload is defined in raw data bytes to be distributed and the throughput rates are calculated in terms of the uncompressed data stream. For constraints inequalities involving the compressed data stream the throughput must be adjusted by the compression factor c. Hardware supported multicast could be modeled in a similar manner. For multicast, the network would be enhanced by newly introduced active switches, but a reliable multicast flow control protocol module must be added at the enpoints and would consume a certain amount of CPU performance and memory system capacity (just like a compression module).

Example: Modelling the switching capacity of an active node for a binary-spanning tree with fast Ethernet and compression.

From the flow chart (similar to Figure 6 but with an additional second output stream from the user buffer to the network) we see two network sends, one network receive, one disk write, four crossings of the I/O bus, eleven streams from and to buffer memory, one compression module and five internal copies of the data stream.

This leads to the following constraints for the maximal achievable throughput b:

$$\begin{aligned}
\tfrac{b}{c} &< 12.5 \text{ MB/s} && \text{link for receive}\\
\tfrac{2b}{c} &< 12.5 \text{ MB/s} && \text{link for send}\\
b &< 24 \text{ MB/s} && \text{SCSI Disk}\\
\tfrac{3b}{c} + b &< 132 \text{ MB/s} && \text{i/o bus, PCI}\\
\tfrac{8b}{c} + 3b &< 180 \text{ MB/s} && \text{memory system}\\
(\tfrac{3}{45c} + \tfrac{1}{80} + \tfrac{4}{90c} + \tfrac{1}{90} + \tfrac{c}{9})b &< 1\ (100\%) && \text{CPU utilization}
\end{aligned}$$

For a compression factor of $c = 2$, an active node in this configuration can handle 5.25 MB/s. The limiting resource is the CPU utilization.

3 Differences in the Implementations

Our first approach to partition-cast was to use a simple file sharing service like NFS[4] with transfers over a UDP/IP network resulting in a *star* topology. The NFS server exports partition images to all the clients in the cluster. A command line script reads the images from a network mounted drive, possibly decompresses the data and writes it to the raw partitions of the local disk. Because of the asymmetric role of one server and many clients, this approach does not scale well, since the single high speed Gigabit Ethernet can be saturated even serving a small number of clients (see performance numbers in Section 4). Although this approach might look a bit naive to an experienced system architect, it is simple, highly robust and supported by every operating system. A single failing client or a congested network can be easily dealt with.

As a second setup, we considered putting together active-clients in a *n-ary spanning tree* topology. This method works with the standard TCP point-to-point connections and uses the excellent switching capability of the Gigabit Ethernet switch backplane. A partition cloning program (called *Dolly*) runs on each active node. A simple server program reads the data from disk on the image server and sends the stream over a TCP connection to the first few clients. The clients receive the stream, write the data to the local disk and send it on to the next clients in the tree. The machines are connected in an n-ary spanning tree, eliminating the bottleneck of the server link accessing the network.

Finally for the third optimal solution the same Dolly client program can be used with a local copy to disk and just one further client to serve. The topology turns into a highly degraded unary spanning tree. We call this logical network a *multi-drop chain*.

An obvious topological alternative would be a true physical spanning tree using the multicasting feature of the networking hardware. With this option the server would only source one stream and the clients would only sink a single stream. The protocols and schemes required for reliable and robust multicast are neither trivial to implement

[4] Network File System

nor included in common commodity operating systems and often depend on the multicast capabilities of the network hardware. In a previous study [11] one of the authors implemented several well known approaches ([6, 10]). Unfortunately the performance reached in those implementation was not high enough to make an application to the partition cloning tool worthwhile.

4 Evaluation of Partition-Cast

In this section we provide measurements of partition casts in different logical topologies (as described in Section 2) with compressed and uncompressed partition images. The partition to be distributed to the target machines is a 2 GByte Windows NT partition. The compressed image file is about 1 GByte in size, resulting in a compression factor of 2.

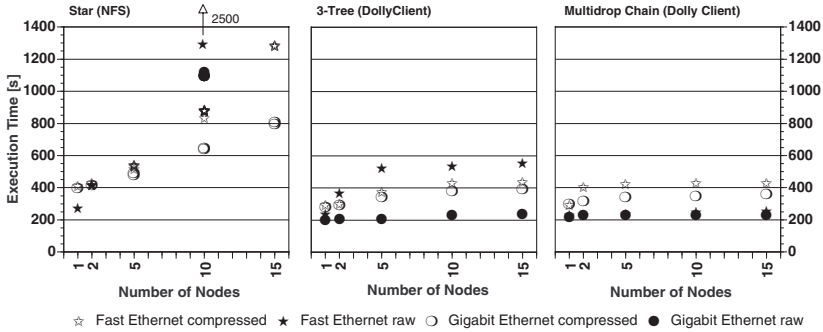


Fig. 7. Total execution times for distributing a 2 GByte Windows NT Operating System partition simultaneously to 1, 2, 5, 10, 15 machines by partition cloning with NFS based star topology, the Dolly based 3-tree and multi-drop-chain topologies on the Patagonia cluster. The star topology run with 10 clients using raw transfers over Fast Ethernet resulted in execution times around 2500 seconds as the NFS servers disk started thrashing.

A first version of our partition-cast tool uses only existing OS services and therefore applies a simplistic star topology approach. It consists of a NFS server which exported the partition images to all the clients. The clients accessed the images over the network using NFS, possibly uncompressing the images and finally writing the data to the target partition. The results of this experiment are shown on the left side of Figure 7 (the execution time for each client machine is logged to show the variability due to congestion). The Figure shows two essential results: (1) The more clients need to receive the data, the more time the distribution takes (resulting in a lower total bandwidth of the system). The bandwidth is limited by the edge capacity of the server. (2) Compression helps to increase the bandwidth for a star topology. As the edge capacity is the limiting factor, the nodes have enough CPU, memory and IO capacity left to uncompress the incoming stream at full speed, thereby increasing the total bandwidth of the channel.

A second approach is to use an n-ary spanning tree structure. This topology was implemented in the small program *Dolly* which acts as an active node. The program reads the partition image on the server and sends it to the first n clients. The clients write the incoming data to the target partition on disk and send the data to the next clients. The out-degree is the same for all nodes (if there are enough successor-nodes) and can be specified at runtime. The results for a 3-ary tree are shown in Figure 7 in the middle. For Fast Ethernet the execution time increases rapidly for a small number of clients until the number of clients (and therefore the number of successor-nodes of the server) reaches the out-degree. As soon as the number of clients is larger than the out-degree, the execution times stay roughly constant. For this network speed, the edge capacity is again the bottleneck, resulting in increasing execution times for higher out-degree. In the case of Gigabit Ethernet, the link speed (the edge capacity) is high enough to satisfy an out-degree of up to 5 without the edge capacity becoming the bottleneck. The bottleneck in this case is still the nodes memory capacity.

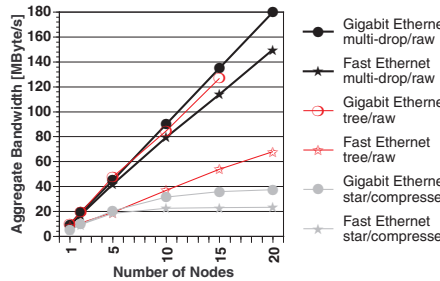


Fig. 8. Total (aggregate) transfer bandwidth achieved in distributing a 2 GByte Windows NT Operating System partition simultaneously to a number of hosts by partition cloning in the Patagonia cluster.

For the third experiment we use Dolly to cast the partition using a multi-drop chain. The results are shown in Figure 7 on the right. They indicate that the execution time for this partition cast is nearly independent on the number of clients. This independence follows from the fact that with the multi-drop chain configuration, the edge capacity is no longer a bottleneck as every edge carries at most one stream. The new bottleneck is the nodes' memory system. The memory bottleneck also explains why compression results in a lower bandwidth for the channel (decompressing data requires more memory copy operations in the pipes to the gunzip process).

Figure 8 shows the total, aggregate bandwidth of data transfers to all disk drives in the system with the three experiments. The figure indicates that the aggregate bandwidth of the NFS-approach increases only modestly with the number of clients while the multi-drop chain scales perfectly. The 3-ary-tree approach also scales perfectly, but increases at a lower rate. The numbers for the NFS approach clearly max out with the transfer bandwidth of the servers network interface reaching the edge capacity: Our NFS-server can deliver a maximum of about 20 MByte/s over Gigabit Ethernet and

about 10 MByte/s over Fast Ethernet (note that we are using compressed data for the NFS approach in the above figure thereby doubling the bandwidth). The predicted bandwidths are compared with measured values in our cluster in Table 1.

Network			Fast Ethernet Bandwidth		Gigabit Ethernet Bandwidth	
Topology	Out-Degree	Com-pression	Extended Model	Measured	Extended Model	Measured
Multi-Drop-Chain	1	no	11.1	8.8	11.1	9.0
Multi-Drop-Chain	1	yes	6.1	4.9	6.1	6.1
2-Tree	2	no	6.3	5.4	8.1	8.2
3-Tree	3	no	4.2	3.8	6.4	8.0
Star	5	no	2.5	2.3	5.3	6.3
Star	5	yes	5.0	3.6	5.0	4.1

Table 1. Predicted and measured bandwidths for a partition cast over a logical chain and different tree topologies for uncompressed and compressed images. All values are given in MByte/s.

5 Conclusion

In this paper we investigated the problem of a partition-cast in clusters of PCs. We showed that optimizing a partition-cast or any distribution of a large block of raw data leads to some interesting tradeoffs between network parameters and node parameters.

In a simple analytical model we captured the network parameters (link speed and topology) as well as the basic processor resources (memory system, CPU, I/O bus bandwidth) at the intermediate nodes that are forwarding our multicast streams. The calculation of the model for our sample PC cluster pointed towards an optimal solution using uncompressed streams of raw data, forwarded along a linear multi-drop chain embedded into the Gigabit Ethernet. The optimal configuration was limited by the CPU performance in the nodes and its performance was correctly predicted at about one third of the maximal disk speed. The alternative of a star topology with one server and 24 clients suffered from heavy link congestion at the server link, while the different n-ary spanning tree solutions were slower due to the resource limitations in the intermediate nodes, that could not replicate the data into multiple streams efficiently enough. Compression resulted in a lower network utilization but was slower due to the higher CPU utilization. The existing protocols for reliable multicast on top of unreliable best-effort hardware broadcast in the Ethernet switch were not fast enough to keep up with our multi-drop solution using simple, reliable TCP/IP connections.

The resulting partition casting tool is capable of transferring a 2 GByte Windows NT operating system installation to 24 workstations in less than 5 minutes while transferring data at a sustained rate of about 9 MByte/s per node. Fast partition cast permits the distribution of entire installations in a short time, adding flexibility to a cluster of PCs to do different tasks at different times. A setup for efficient multicast also results

in easier maintenance and enhances the robustness against slowly degrading software installations in a PC cluster.

References

[1] Henri Bal. The Distributed ASCI Supercomputer (DAS). *http://www.cs.vu.nl/~bal/das.html*.

[2] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *Proceedings of 1995 ICPP Workshop on Challenges for Parallel Processing*, Oconomowc, Wisconsin, U.S.A., August 1995. CRC Press.

[3] Nanette J. Boden, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet — A Gigabit per Second Local Area Network. *IEEE-Micro*, 15(1):29–36, February 1995.

[4] William J. Bolosky, Joseph S. Barrera III, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid. The Tiger Video Fileserver. In *Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, Zushi, Japan, April 1996. IEEE Computer Society.

[5] Dolphin Interconnect Solutions. *PCI SCI Cluster Adapter Specification*, 1996.

[6] S. Floyd, V. Jacobson, S. McCanne, L. Zhang, and C-G. Liu. A Reliable Multicast Framework For Lightweight Sessions and Application Level Framing. In *Proceedings of ACM SIGCOMM '95*, pages 342–356, August 1995.

[7] H. Hellwagner and A. Reinefeld, editors. *SCI Based Cluster Computing*. Springer, Berlin, Spring 1999.

[8] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O Malley. Logical vs. Physical File System Backup. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, Louisiana*, pages 239–249. The USENIX Association, February 1999.

[9] Steve Kotsopoulos and Jeremy Cooperstock. Why Use a Fishing Line When you Have a Net? An Adaptive Multicast Data Distribution Protocol. In *Proccedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996. The USENIX Association.

[10] Sanjoy Paul, Krishan K. Sabnani, and David M. Kristol. Multicast Transport protocols for High Speed Networks. In *Proceedings of International Conference on Network Protocols*, pages 4–14. IEEE Computer Society Press, 1994.

[11] F. Rauch. Zuverlässiges Multicastprotokoll. Master's thesis, ETH Zürich, 1997. English title: Reliable Multicast Protocol. See also http://www.cs.inf.ethz.ch/. Contains a survey about reliable IP multicast.

[12] Felix Rauch, Christian Kurmann, Thomas Stricker, and Blanca Maria Müller. Patagonia — A Dual Use Cluster of PCs for Computation and Education. In *2. Workshop Cluster Computing, Karlsruhe*, March 1999.

[13] Rich Seifert. *Gigabit Ethernet: Technology and Applications for High-Speed LANs*. Addison-Wesley, May 1998. ISBN: 0201185539.

[14] T. Stricker and T. Gross. Optimizing Memory System Performance for Communication in Parallel Computers. In *Proc. 22nd Intl. Symposium on Computer Architecture*, pages 308–319, Santa Marguerita di Ligure, June 1995. ACM.

A New Home-Based Software DSM Protocol for SMP Clusters*

Weiwu Hu, Fuxin Zhang, and Haiming Liu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing 100080
hww@water.chpc.ict.ac.cn

Abstract. This paper introduces an SMP protocol for the home-based software DSM system JIAJIA. In the protocol, intra-node processes in an SMP node share their home pages through hardware coherent sharing so as to take the full advantage of the *home effect* of home-based software DSMs. In contrast, cached remote pages of a process are not shared by its intra-node partners to avoid cache page conflict within an SMP. Besides, JIAJIA also implements the shared memory communication among processes within the same SMP node to accelerate intra-node communication. Performance evaluation with some well accepted benchmarks and real applications in a cluster of four two-processor nodes shows that the SMP protocol of JIAJIA reduces remote accesses, diffs, and consequently message amounts in all of the ten benchmarks and as a result obtains noticeable performance improvement in seven.

1 Introduction

With the wide spread of symmetric multiprocessor (SMP) systems, cluster of SMPs has been emerging as an attractive parallel processing platform to provide high performance with certain connectivity and affordable costs. Given the convenient shared address programming model of SMPs, it is natural to extend this convenience to the SMP clusters with shared virtual memory. The shared virtual memory has the obvious advantages over the message passing alternative on SMP clusters in that it can take the advantage of the efficient hardware-based shared memory within an SMP. In a software DSM on cluster of SMPs, the SMP hardware transparently provides shared memory at the cache line granularity for intra-node sharing, while the software protocol is responsible for providing shared memory at the page granularity for inter-node sharing.

Previous software DSMs on SMP clusters include the softFLASH system[3] which implements a single-writer protocol for sequential consistency, the Shasta system[12] which performs coherence through code instrumentation at cache-line granularity, the Cashmere-2L[13] which is based on the DEC Memory Channel

* The work of this paper is supported by National Natural Science Foundation of China (Grant No. 69703002) and National High Technology (863) Program (Grant No. 863-306-ZD01-02-2).

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1132–1142, 2000.
© Springer-Verlag Berlin Heidelberg 2000

network interface and network, the HLRC-SMP[11] which is an implementation of a lazy, home-based, multiple-writer protocol across SMP nodes and which uses the Virtual Memory Mapped Communication (VMMC-2) library for the Myrinet network, and the modified version of TreadMarks[7] which uses POSIX threads instead of processes to implement parallelism within a multiprocessor.

This paper introduces design and implementation of a new software DSM protocol for SMP clusters. The protocol is designed and implemented on the home-based software DSM system JIAJIA[4] and hence has a similar goal to the HLRC-SMP protocol. The main difference between HLRC-SMP and the SMP protocol of JIAJIA is that processes in an SMP share both home and cached pages in HLRC-SMP, while only home pages are shared by intra-node processes in JIAJIA. Sharing home pages among intra-node processes in SMP clusters helps to take the full advantage of the *home effect* of home-based software DSMs, i.e., the ability to dispense with page faults for references made by the home node of a given page. Though sharing cached pages is helpful for all processes within a node to benefit from a page fetch performed by one, it also causes cache page conflict such as when a process is writing a cached page, another process in the same node may covers this page due to a page fetch.

Another optimization JIAJIA makes for SMP clusters is the optimization of intra-node communication. With this optimization, processes within an SMP node communicate through shared memory.

The effect of the SMP protocol is evaluated on a Myrinet cluster of four two-processor Ultra-2 nodes with ten benchmarks, include Water, Barnes, Ocean, and LU from SPLASH2, MG and 3DFFT from NAS Parallel Benchmarks, SOR, and TSP from TreadMarks benchmarks, and two real applications EM3D for magnetic field computation and IAP18 for climate simulation. Evaluation results show that the SMP optimization achieves a speedup of $4\% - 5\%$ in LU, SOR, and EM3D, about 10% in MG and 3DFFT, and more than 20% in Ocean and IAP18.

The rest of this paper is organized as follows. The following Section 2 briefly introduces the JIAJIA software DSM system. Section 3 illustrates the SMP protocol of JIAJIA. Section 4 presents experiment results and analysis. The conclusion of this paper is drawn in Section 5.

2 The JIAJIA Software DSM System

In JIAJIA, each shared page has a home node and homes of shared pages are distributed across all nodes. References to home pages hit locally, references to non-home pages cause these pages to be fetched from their home and cached locally. A cached page may be in one of three states: Invalid (INV), Read-Only (RO), and Read-Write (RW). When the number of locally cached pages is larger than the maximum number allowed, some aged cache pages must be replaced to its home to make room for the new page. This allows JIAJIA to support shared memory that is larger than physical memory of one machine.

JIAJIA implements the scope memory consistency model. Multiple writer technique is employed to reduce false sharing. In JIAJIA, the coherence of cached pages is maintained through requiring the lock-releasing (or barrier arriving) processor to send to the lock write notices about modified pages in the associated critical section, and the lock-acquiring (or barrier leaving) processor to invalidate cached pages that are notified as obsolete by the associated write-notices in the lock. This protocol maintains coherence through write notices kept on the lock and consequently eliminates the requirement of directory.

The following optimizations[4,6] are made to the protocol.

Single-Writer Detection. In the optimization, if a page is modified only by its home processor during a synchronization interval, then it is unnecessary to send the associated write notice to the lock manager at the end of the interval. If a page is modified only by one remote processor during an interval, then the processor who makes the modification need not invalidate the page on next acquire or barrier.

Incarnation Number[8] Technique. With this optimization, each lock is associated with an incarnation number which is incremented when the lock is transferred. A processor records the current incarnation number of a lock on an acquire of the lock. When the processor acquires the lock again, it tells the lock manager its current incarnation number of the lock. With this knowledge, the lock manager knows which write notices have been sent to the acquiring processor on previous lock grants and excludes them from write notices sent back to the acquiring processor at this time.

Lazy Home Page Write Detection. Normally, home pages are write-protected at the beginning of a synchronization interval so that writes to home pages can be detected through page faults. The lazy home page write detection delays home page write-protecting until the page is first fetched in the interval so that home pages that are not cached by remote processors do not need to be write-protected.

Write Vector Technique. The write vector optimization is motivated by the idea of fetching diffs only on a page fault in homeless protocols. It avoids fetching the whole page on a page fault through dividing a page into blocks and fetches only those blocks that are dirty with respect to the faulting processor on a page fault. A write vector table is maintained for each shared page in its home to record for each processor which block(s) has been modified since the processor fetched the page last time.

Home Migration. The home migration optimization adaptively migrates home of a page to the processor most frequently writes to the page, to reduce diff overhead at the end of an interval because write to home pages does not produces twin and diff in home-based protocol. In the home migration scheme, pages that are written by only one processor between two barriers are recognized by the barrier manager and their homes are migrated to the single writing processor. Migration information is piggybacked on barrier messages and no additional communication is required for the migration.

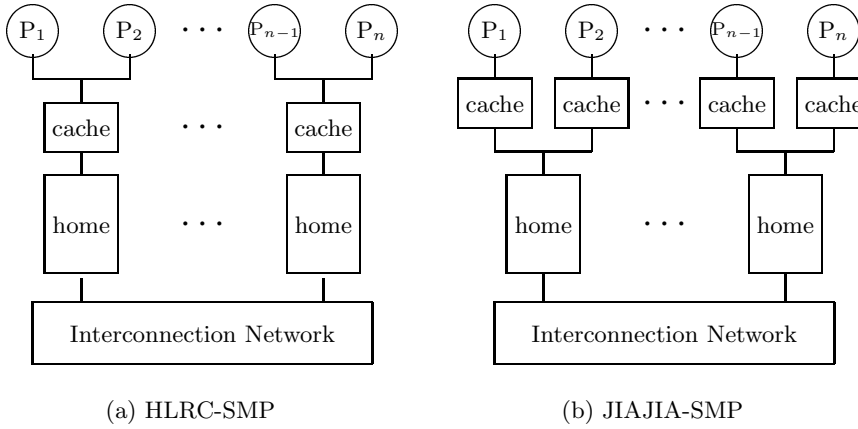


Fig. 1. Memory Organization of HLRC-SMP and JIAJIA-SMP

3 SMP Protocol for JIAJIA

3.1 Design Alternatives

Processors within an SMP can share data either through threads or through processes. The thread model provides an intuitive and efficient method of sharing data within an SMP. However, threads may cause some unexpected side effects. First, all threads within an SMP implicitly share all global variables, which makes it difficult to provide a uniform sharing memory between threads within a node and threads in different nodes. Second, all threads within an SMP have the same view of shared data, which means if a thread decides to invalidate a shared page, the page is invalidated for other threads as well.

The alternative is to use processes within an SMP node to share memory. One approach is to use `shmget()` and `shmat()` system calls. The major disadvantage of this approach is that both the number of shared segments and segment size are limited by the system, preventing it from being used by JIAJIA which can support a large shared memory. Besides, the system overhead of `shmget()` increases linearly with the number of segments[2].

Another and more efficient approach of sharing memory among processes with an SMP is to rely on virtual memory management and the `mmap()` system call (anonymous mapping with `MAP_SHARED` parameter). JIAJIA adopts this approach after comparing the portability, programmability, ability to support large memory, and implementation simplicity of all candidate alternatives. The disadvantage of this approach is that shared pages should be mapped before processes are forked in an SMP node. To meet this requirement, JIAJIA reserves a shared space before processes are forked in an SMP node at the initialization stage.

3.2 SMP Protocol

Figure 1 shows the memory organization of HLRC-SMP and SMP protocol of JIAJIA. As can be seen from Figure 1, the main difference between HLRC-SMP and the SMP protocol of JIAJIA is that processes in an SMP share both home and cached pages in HLRC-SMP, while only home pages are shared by processes in an SMP in JIAJIA.

Home-based software DSMs benefit from the *home effect*, the ability to dispense with page faults for references made by the home node of a given page. In SMP clusters, there are several processes running on an SMP node, each process is the home of part of the total shared pages. Sharing home pages among processes within a node combine their home pages together so that each process in the node "sees" a larger home.

Sharing cached pages within an SMP seems not so attractive to JIAJIA. Though with shared cache a processor may benefit from a remote page fetch carried out by another processor in the same SMP node, shared cache pages cause sharing violation among processors within the same SMP. For example, when a process fetches a page from its home, it has to ensure that no other processes in the same node are currently writing to the page, otherwise the incoming page may overwrite these writes. Besides, to support shared space larger than the physical memory of a machine, a cached page is dynamically mapped and unmapped to the faulting address when the page is cached and flushed by a process in JIAJIA. The dynamic mapping and unmapping of cached pages violates the requirement that pages physically shared by intra-node processes through `mmap()` should be mapped before intra-node processes are forked.

The decision of shared home and separated cache for intra-node processes greatly simplifies the SMP protocol of JIAJIA. Each processor in an SMP maintains cache coherence as if in cluster of single-processor nodes. Sharing home pages among intra-node processes is very simple and requires least modification to JIAJIA. Though all processes share their home pages, each process has its separate view (such as protect state) of a given home page and operates on the page as if it is the unique home host of the page. To service remote page fetch and diff request, one process is appointed as the *real home host* of a given page and a page fetch or diff request is always sent to the real home host of the faulting page. Real home hosts of shared pages in an SMP node are distributed across all processes within the node to avoid bottleneck problem. The process that regards a page as its home page, but does not service remote request about the page, are called *co-home host* of the page.

Though the SMP protocol of JIAJIA is simplified in the absence of some intra-node processes cache sharing related complexity, non-trivial issues still arise when coordinating with some optimization methods of JIAJIA.

In the lazy home page write detection, the home page write-protecting that are performed at the beginning of an interval is delayed until the page is first fetched in the interval so that home pages that are not cached by remote processors do not need to be write-protected. In the SMP protocol, a page fetch request is only sent to the real home host of the page, co-home hosts do not

know when the page is first fetched at an interval. Therefor, the lazy home page write detection technique is applied only to the real home host of a given page, and the co-home hosts of a page write-protect the page at the beginning of each interval as normal. With the observation that only the process that modifies a page needs to detect the write, the real home host of a given page can be dynamically shifted to the process that write the page most frequently.

In the write vector technique, a home page is divided into blocks and a write vector table is maintained for each shared page in its home to record for each processor which block(s) has been modified since the processor fetched the page last time, so that only modified blocks are sent back when the processor requires the page. In the normal protocol, updates by remote processors to a page are recorded in the write vector table when diffs are applied, and updates by the home node are recorded with the twin mechanism. In the SMP protocl, the modifications made by the co-home hosts of a page should also be known by the real home host. JIAJIA solves this problem through making the write vector table of any given page shared among all processes within a node, so that modifications to the write vector table made by co-home hosts are always visible to the real home host.

The home migration technique also causes problem to the SMP protocol of JIAJIA. As has been mentioned, the `mmap()` mechanism JIAJIA uses to share physical memory among intra-node processes requires that shared pages are mapped before intra-node processes are forked. In contrast, the home migration algorithm of JIAJIA maps and unmaps pages dynamically during runtime. Compromise has to be made here that if the home of a shared page is migrated from one node to another, then the advantage of sharing home page among intra-node processes is given up for this page. The new home host is the real home host of the page, no co-home host exists for this page.

3.3 Intra-node Communication

Another optimization JIAJIA makes for SMP clusters is the optimization of intra-node communication. In the old version of JIAJIA which regards processors of an SMP as separate nodes, UDP protocol is used for communication between two intra-node processes. With the SMP communication support, processes within an SMP node communicate through shared memory.

To support shared memory intra-node communication, JIAJIA allocates a shared communication buffer for each pair of intra-node processes. When a process wants to send a message to another process within the same node, the sending process first copies the message to the associated shared buffer. It then sends a SIGIO signal to the receiving process through `kill()` system call. On receiving the SIGIO signal, the receiving process directly reads the message from the shared communication buffer and then sets a tag in the buffer to indicate that the message has been read out. The sending process proceeds on observing this receiving tag.

To keep the simplicity of implementation, the current version of JIAJIA does not include some other SMP-related optimizations such as intra-node synchro-

Table 1. Characteristics and Sequential Run time of Benchmarks

Appl.	Size	Mem.	Barrs	Locks	Seq. Time(seconds)
Water	1728 mole.	0.5MB	70	1040	491.99
Barnes	16384 bodies	1.6MB	28	64	321.83
Ocean	$514 * 514$	60MB	858	1568	54.88
LU	$2048 * 2048$	32MB	128	0	133.45
SOR	$2048 * 2048$	16MB	200	0	89.44
TSP	`-f20 -r15`	0.8MB	0	1167	216.70
MG	$256 * 256 * 256$	443MB	592	0	398.56*
3DFFT	$128 * 128 * 128$	96MB	12	0	74.05
EM3D	$120 * 60 * 416$	160MB	20	0	101.46
IAP18	$144 * 91 * 18$	20MB	5400	0	1994.95

$*$: Estimated from $128 \times 128 \times 128$ MG sequential time (49.82 seconds). Sequential time of $256 \times 256 \times 256$ MG is unavailable due to memory size limitation.

Table 2. Runtime Statistics of Parallel Execution

Appl.	Msg. amt.(MB)		Get pages		Diffs	
	JIA	JIA_{smp}	JIA	JIA_{smp}	JIA	JIA_{smp}
Water	36	29	3420	2870	1897	1373
Barnes	81	65	9076	7408	510	353
Ocean	891	563	107073	67733	1294	1132
LU	25	18	12072	8292	0	0
SOR	12	5.3	2823	1216	0	0
TSP	7.4	6.3	4965	4237	3236	2780
MG	553	260	32508	18025	43192	17016
3DFFT	149	129	17976	15408	56	48
EM3D	2.4	1.1	1200	528	990	420
IAP18	2943	1355	276643	136474	131136	49920

nization and locating fault pages in the caches of intra-node processes on page faults.

4 Performance Evaluation

The evaluation is done in four Ultra-2 nodes connected by a Myrinet. Each node has two 200MHz Ultra-sparc processor and 256MB memory.

The benchmarks include Water, Barnes, Ocean and LU from SPLASH2[14], MG and 3DFFT from NAS Parallel Benchmarks[1], SOR and TSP from TreadMarks benchmarks[10], and two real applications EM3D for magnetic field computation and IAP18 for climate simulation[5]. Table 1 gives characteristics and sequential run time of these benchmarks.

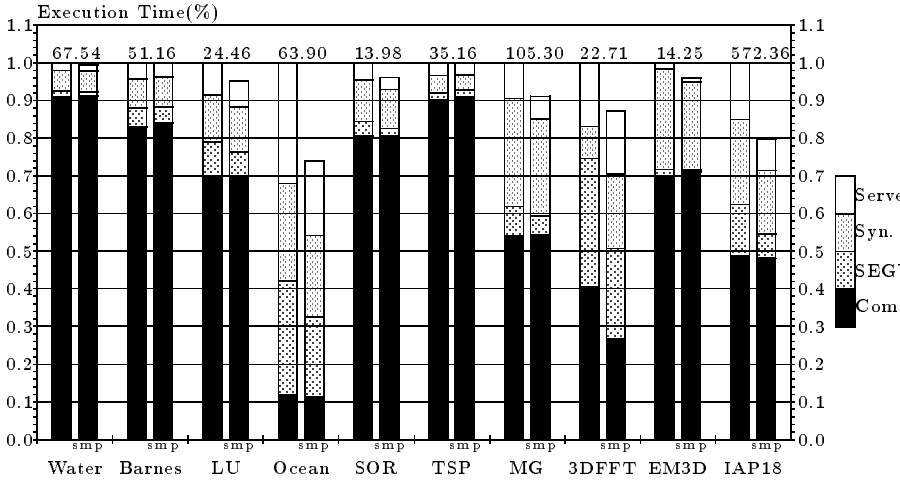


Fig. 2. Parallel Execution Time Breakdown

Each benchmark is run under JIAJIA both with (JIA_{smp}) and without (JIA) SMP optimization. The four nodes of two-processor SMPs are configured as eight separate hosts in JIA, and as four SMPs in JIA_{smp}.

Table 2 gives some statistics of the parallel execution. Message amount, remote get page request count, and diff count are listed in Table 2 for each run of each benchmark. Figure 2 shows parallel execution time results of the execution. For each benchmark, the parallel execution time of JIA rides on the top of the corresponding bar, and the parallel execution time of JIA_{smp} is represented as percentages of the parallel execution time of JIA. In Figure 2, the execution time of each parallel run is broken down into four parts: page fault (SIGSEGV service) time, synchronization time, remote request (SIGIO) service time, and computation time. The first three parts of time are collected at runtime as the overhead of the execution, and the computation time is calculated as the difference of the total execution time and the total overhead.

It can be seen from Table 2 and Figure 2 that, all of the ten benchmarks have less remote accesses, diffs, and consequently message amounts in JIA_{smp} than in JIA and seven of them achieve noticeable speedups with the SMP protocol.

Figure 2 shows that JIA_{smp} achieves significant speedup over JIA in Ocean (26.0%), MG (8.9%), 3DFFT (12.9%), and IAP18 (20.3%). The speedup of Ocean turns from negative in JIA to positive in JIA_{smp}. The common property of these four benchmarks is that their parallel speedup are not high. The eight-processor speedup of JIA is negative in Ocean, and is less than four in MG, 3DFFT, and IAP18. Therefore there is large room for performance improvement. Besides, the absolute number of remote accesses, diffs, and consequently message amounts reduced by the SMP protocol is great in Ocean, MG, and IAP18. Table 2 shows that JIA_{smp} causes 300MB less message amounts than JIA in Ocean

and MG, and 1.6GB less message amounts than JIA in IAP18. As a result of the great reduction of page faults, both the SIGSEGV time and server time are significantly reduced, as can be observed in Figure 2. The synchronization time of Ocean, MG, and IAP18 are also reduced due to the reduction of the number of diffs (diffs are generated at synchronization points). In FFT, though the message amount reduced by the SMP protocol is not as significant as the above three benchmarks, JIA_{smp} still reduces SIGSEGV time significantly. It can also be seen from Figure 2 that a great part of performance gains of JIA_{smp} over JIA in FFT is caused by the reduction of computation time in JIA_{smp}. This fact implies that FFT has better utilization of processor pipeline, cache, and TLB in JIA_{smp} than in JIA.

Figure 2 also shows that LU, SOR, and EM3D obtain moderate performance benefit (around 5%) from the SMP protocol. It can be seen from Table 2 that, though JIA_{smp} has only 43%, 44%, and 69% remote accesses of that required by JIA in SOR, EM3D, and LU, the absolute message amounts reduced are not significant. Besides, the performance of these three benchmarks is already high in JIA (the eight-processor speedup is 5.46 in LU, 6.4 in SOR, and 7.12 in EM3D) and the room for performance improvement is not large. Again, the execution time breakdown in Figure 2 shows that in LU, SOR, and EM3D, the performance gains of JIA_{smp} over JIA is mainly caused by the reduction of SIGSEGV time and server time as a result of remote accesses reduction. In EM3D, the synchronization time is also reduced in JIA_{smp} because JIA_{smp} generates much less diffs than JIA.

In Water, Barnes, and TSP, though the number of remote accesses and diffs is reduced with the SMP protocol, both the reduction range and the absolute message amounts reduced are not large. Besides, in these three benchmarks, performance is already high in JIA (the eight-processor speedups is 7.28 in Water, 6.29 in Barnes, and 6.16 in TSP) and the bottleneck for further performance improvement does not lie in the communication. Rather, synchronization is the major obstacle for further performance improvement in these three benchmarks. It can be seen from Table 1 and Figure 2 that lock is the major synchronization mechanism and synchronization overhead constitutes the major overhead in these three benchmarks. This implies that the lock waiting time dominates the overhead and (with the high bandwidth Myrinet) the moderate remote accesses reduction caused by JIA_{smp} has little influence on the performance of Water, Barnes, and TSP.

5 Conclusion and Future Work

This paper introduces the SMP protocol for the JIAJIA software DSM system. In the protocol, processes in the same SMP node keep their separate caches but combine their home pages together so that each process in the node "sees" a larger home.

Evaluation results show that compared to the JIAJIA without SMP optimization, the SMP protocol achieves significant speedups in seven out of ten

benchmarks. Quantitatively, the SMP protocol achieves a speedup of $4\% - 5\%$ in LU, SOR, and EM3D, about 10% in MG and 3DFFT, and more than 20% in Ocean and IAP18. Water, Barnes, and TSP do not benefit noticeably from the SMP protocol. It is expected that the SMP protocol of JIAJIA can perform better if there are more processors (e.g., four) within an SMP node.

It can also be learned from the above performance evaluation that communication bandwidth is critical to the performance of software DSMs such as JIAJIA. In the evaluation, the high bandwidth Myrinet makes JIAJIA not sensitive to the moderate reduction of remote accesses in Water, Barnes, and TSP. Compared to our previous evaluation results[4,6] on clusters connected by 100Mbps Ethernet, higher speedups are obtained in the cluster connected by Myrinet.

Our recent work about JIAJIA include further improving the coherence protocol of JIAJIA, supporting JIAJIA with faster communication mechanism such as special remote access hardware, implementing fault tolerance mechanism such as checkpoint in JIAJIA, building a pre-compiler to convert OpenMP programs to the API of JIAJIA, and porting more real applications on JIAJIA. Further information about JIAJIA is available at `www.ict.ac.cn/chpc/index.html`.

References

1. D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks", *Technical Report 103863*, NASA, Jul. 1993.
2. S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets, "Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory", in *Proc. of the 13th Int'l Parallel Processing Sym.*, pp. 153–159, April. 1999.
3. A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory", in *Proc. of the 996 Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
4. W. Hu, W. Shi, and Z. Tang, "Reducing System Overhead in Home-Based Software DSMs", in *Proc. of the 13th Int'l Parallel Processing Symp.*, pp. 167–173, Apr. 1999.
5. W. Hu, F. Zhang, L. Ren, W. Shi, and Z. Tang, "Running Real Applications on Software DSMs", in *Proc. of the 2000 Int'l Conf. on High Performance Computing in the Asia-Pacific Region*, May 2000.
6. W. Hu, "Reducing Message Overheads in Home-Based Software DSMs", in *Proc. of the 1st Workshop on Software Distributed Shared Memory*, pp. 7–11. June 1999.
7. Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs", in *Proc. of the 13th Int'l Parallel Processing Symp.*, pp. 302–310. Apr., 1999.
8. L. Iftode, "Home-based Shared Virtual Memory", *Ph. D. Thesis*, Princeton University, Aug. 1998.
9. P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "TreadMarks Distributed Shared Memory on Standard Workstations and Operating Systems", in *Proc. of the 1994 Winter Usenix Conf.*, pp. 115–131, Jan. 1994.
10. H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel, "Quantifying the Performance Differences Between PVM and TreadMarks", *Journal of Parallel and Distributed Computing*, Vol. 43, No. 2, pp. 65–78, Jun. 1997.

11. R. Samanta, A. Bilas, L. Iftode, and J. Singh, "Home-based SVM Protocols for SMP Clusters: Design and Performance", in *Proc. of the 4th Int'l Symp. on High Performance Computer Architecture*, Feb., 1998.
12. D. Scales, K. Gharachorloo, and A. Aggarwal, "Fine-grain Software Distributed Shared Memory on SMP Clusters", in *Proc. of the 4th Int'l Symp. on High Performance Computer Architecture*, pp. 125–136, Feb., 1998.
13. R. Stets et al., "Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network", in *Proc. of the 1997 ACM Symp. on Operating Systems Principles*, pp. 170–183, Oct. 1997.
14. S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", in *Proc. of ISCA'95*, pp. 24–36, 1995.

Encouraging the Unexpected: Cluster Management for OS and Systems Research

Ronan Cunniffe and Brian A. Coghlan

Department of Computer Science
Trinity College Dublin, Ireland
{ronan.cunniffe, brian.coghlan} @cs.tcd.ie

Abstract. A framework for cluster management is proposed that enables a cluster to be more efficiently utilized within a research environment. It does so by removing cluster management to a management node, leaving the compute nodes as essentially bare machinery. Users may schedule access to one or more of the compute nodes via the management node. At the scheduled time, a previously-saved image of their research environment is loaded, and the session begun. At the end of the session the user may save a new image of the environment on the management node, to be reloaded at another time. Thus the user may work with a customized environment, which may even be a fledgling operating system, without fear of interference to other researchers. This enables the capital investment of a systems research cluster to be amortized over a greater number of researchers.

1. Introduction

We assume clusters represent a significant capital investment, and that in order to maximise the utilization of that investment they are very closely managed, with jobs scheduled on the compute nodes by management software such as CCS [1], PBS [2,3], LSF[4] or LoadLeveller [5] integrated with the working environment. It is likely then, that any modifications to the working environment that might jeapordize stability will be very unwelcome.

Hence OS or systems research, that is likely to jeopardize stability, requires either a private cluster or diplomatic negotiations over how far any researcher can modify the shared working environment. The former is a very inefficient use of funding, the latter a severe constraint on research targets. Here we propose a scheme that allows the researcher the illusion of a private cluster whilst in fact using a shared cluster.

2. The MultiOS Framework

What we wish to describe here is the general philosophy rather than a specific implementation, since it is felt that any implementation may need to be adapted to site-specific tools (schedulers, for instance) and the adaptability should be part of the

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1143-1147, 2000.
© Springer-Verlag Berlin Heidelberg 2000

philosophy. However, describing the framework is probably best done in concrete terms, by describing the cluster for which it is being initially designed, how it will operate on that cluster, and what hardware/software tools are required.

Our cluster is made up of sixteen PC compute nodes and two storage servers, linked by a switched fabric of SCI links. In addition, there are two NIS servers, a http server and a firewall server. The compute nodes are arranged in logical groups of 4. Each compute node has 256MB of DRAM and a 2GB local hard-disk. The storage servers are connected to a large RAID, and all machines are connected to the external network via 100Mb/s Ethernet. All normal access to the cluster is through this network connection and physical access to compute nodes and servers is minimised. The MultiOS server will execute either on an extra server node, or on one of the existing servers. Over time, it is hoped to increase the number of compute nodes.

The central idea behind the MultiOS framework is that between two successive zero-management 'research' sessions there is a 'management' session where *all* access is suspended, and the environment installed on the local hard-disks can be changed by management software according to the schedule or via user interaction with the MultiOS server. Three requirements must be met to do this. The MultiOS server must be able to:

1) *force* a reboot of any compute node on demand (via a hardware mechanism)
2) gain control of a compute node during boot - before any environment starts.
3) install and run a management environment that does not use the local disk.

2.1 A Hardware Reset Mechanism

There is no guarantee that a running environment will respond gracefully to a request to shut down, indeed the highly experimental work this framework is designed to support is quite likely to crash or lock up the hardware it is running on. Disruption of the schedule is not acceptable, nor is demanding human intervention. MultiOS *must* be able to regain control after a crash. In our case, this will either be implemented using a LonWorks network [6] with the module in each compute node wired to the reset pin, or a parallel switched 100Mbps Ethernet fabric with modified wake-on-LAN [7].

2.2 Control Must Be Passed to the MultiOS Server During Boot

This can be done by using the standard protocols for booting diskless workstations but in a slightly non-standard way. Most PC Ethernet cards have a socket for a 'bootrom', which can be recognised by the normal boot sequence and invoked before any other bootable device. This bootrom sends a BOOTP [8] request to find the machine's own identity and the name of a file to download, and then uses TFTP [9] to download it. Once downloaded, that file is executed.

Obviously, if two different files are downloaded on two successive boots, the compute node will boot differently. This is what MultiOS does, alternating between two executables. The first executable is the management environment; the second is a tiny program which simply passes control straight on to the boot-block of the local

hard-disk, so that the compute node boots into whatever target environment has been installed, as though the network interrogation phase never occurred.

2.3 A Special Management Environment Which Does Not Use the Local Disk

This special management environment could be a specialised program, or a suite of specialised programs. Many already exist [10, 11, 12], but hardcoded solutions are not flexible - it is difficult to take advantage of a hardware configuration or network topology unless the program is re-written for each specific cluster. It also does not easily exploit possible redundancy of transmission, where multiple compute nodes are to be loaded with near-identical images. Essentially this approach suffers from conception in a vacuum, where every new optimisation requires new tools or low-level modifications to existing ones.

A much more flexible and powerful approach is to use a fully featured operating system, and to assemble custom solutions from its native toolkit. In our case this OS is Linux - the management environment downloaded via TFTP is actually a Linux kernel configured as for a diskless workstation - mounting some of its filesystems in memory, and the remainder over NFS (the OS images for instance). The high-level tools for moving OS images across the network are then built on the standard UNIX commandline tools, such as *dd, gzip, diff, rcp*, etc. The main costs of using Linux are the image storage (a 'minimal' Red Hat 6.0 install, slightly modified for diskless operation, is 20MB per node and 80MB shared by all), and the network bandwidth consumed by multiple parallel accesses to this resource.

3. The MultiOS Server

We have described the low-level mechanism for transparently switching between research environments. A permanent high-level control mechanism - the MultiOS server - is also needed, to implement user commands and to provide status information. A further requirement is to allow users to reserve some or all of the cluster in advance.

In line with our preferences for adaptability, it is proposed not to integrate a custom node reservation system or scheduler into the MultiOS server, but to use an external program. A basic implementation will be provided, but a modular approach allows it to be easily replaced. Likewise, we do not intend to integrate a user interface into this server, allowing for whatever variations on user access are desired. We intend to provide a web-based console, but this may not always be appropriate. The overall structure of the MultiOS ‘server’, therefore, is really a set of three elements: the user interface program, the reservation system, and the actual MultiOS server program that controls the low-level operation of the framework.

4. Security Issues

The MultiOS scheme introduces important security issues. Four primary areas of concern have been identified.

Firstly, the BOOTP and TFTP protocols were designed to be small rather than secure, and are vulnerable to a variety of attacks, denial-of-service (DOS), IP and MAC address spoofing, etc. A successful attack results in the node running an OS of the attacker's choice. Machines booting in this way have to be considered as totally unprotected, and therefore a weak point in whatever network(s) they belong to. The only effective workaround is to boot from a separate physically secure network.

Secondly, the image server must allow external access to the environment images, both for users and for the MultiOS framework, and both read and write access must be subject to authorization. However, since a management cycle involves access from the compute nodes, the user's authorization must be transmitted explicitly to them. Transmissions can be encrypted, but transport layer security is useless if one of the parties is compromised. Again, using a switched private network eliminates this concern.

Thirdly, it is ironic that the behaviour of a cluster being reset is indistinguishable from a well-synchronised distributed-denial-of-service (D-DOS) attack on the MultiOS server, mounted by a large number of high-bandwidth attackers. Services such as BOOTP, TFTP, and the network filesystems NFS and SMB are commonly configured to shut themselves down when attacked, rather than overload the server machine they run on. For the purposes of MultiOS, either the cluster-resetting must be done in staggered fashion to keep the load below the triggering threshold, or this mechanism must be disabled, and the cluster made physically secure against such attacks instead. Since the MultiOS server directly controls the compute node reset mechanism, the former is likely to be easier.

The final area of security risk is the user interface. A web-based interface is favoured for customisability, platform independence and ease-of-use, but hard experience teaches that web servers can be crashed. This is a secondary argument for splitting the user-interface away from the MultiOS server. It can now reside on a separate machine, communicating through an encrypted channel. In this scenario, a webserver crash will shut down interactive access to MultiOS and prevent changes being made to the schedule, but will not affect the cluster itself.

5. Summary

A framework for cluster management has been proposed that is specifically tailored to OS and systems software research. Users may schedule access to one or more of the compute nodes via a separate management node, the MultiOS server. At the appointed time, a previously-saved image of their research environment will be loaded from an image storage server, and the session begun. At any time during the session the user may save a new image of the environment, to be reloaded at another time. Hence a number of users can be accommodated, each within their own environment. This

enables the capital investment of the cluster to be amortized over a greater number of researchers.

Work on the framework, called MultiOS, began in October, 1999, and is still in progress. Our thanks to Prof. J. G. Byrne for his support.

References

1. Keller, A., Reinefeld, A., "CCS Resource Management in Networked HPC Systems", Proc. Heterogeneous Computing Workshop HCW'98, 1998

2. Portable Batch System Documentation, MRJ Ltd., 1998. http://pbs.mrj.com/docs/html

3. Henderson, R.L., "Job Scheduling under the Portable Batch System, In: Job Scheduling Strategies for Parallel Processing", Feitelson, D.G. and Rudolph, L. (eds), LNCS, pp.279-294, Vol.949, Springer-Verlag, 1995.

4. Load Sharing Facility Suite 3.2 Documentation, Platform Computing Inc., 1998.

5. Prennis, A.jnr, "Loadleveller: workload management for parallel and distributed computing environments", Proc. Supercomputing Europe (SUPEUR'96), October 1996.

6. Foster, G.T., Glover, J.P.N., Warwick, K., "Flexible Distributed Control of Manufacturing Systems Using Local Operating Networks", Proc. LonUsers International Fall Conference, 1995

7. Magic Packet Technology White Paper, AMD Publication no.20213, Advanced Micro Devices Inc., November 1995

8. Wimer, W., "Clarifications and Extensions for the Bootstrap Protocol", IETF Request For Comments Document no.1542, October 1993.

9. Sollins, K., "The TFTP Protocol (Revision 2)", IETF Request For Comments Document no.1350, July 1992.

10. Rembo Technology, http://www.bpbatch.org

11. Free Software Foundation, "Grand Unified Bootloader" http://www.gnu.org/software/grub.en.html

12. Yap, K; Savoye, R; "Network Interface Loader" http://nilo.sourcefourge.net/

Flow Control in ServerNet^R Clusters

Vladimir Shurbanov[1], Dimiter Avresky[1], Pankaj Mehra[2], and William Watson[3]

[1] Boston University, 8 Saint Mary's St., Boston, MA 02215, USA,
{vash,avresky}@bu.edu
[2] Compaq Tandem Labs, 19333 Vallco Parkway, Cupertino, CA 95014
pankaj.mehra@compaq.com,
[3] Compaq Tandem Labs, 14231 Tandem Blvd., Austin, TX 77728
william.watson@compaq.com

Abstract. This paper investigates the performance implications of several end-to-end flow-control schemes based on the ServerNet^R system-area network. The static window (SW), packet pair (PP), and the simplified packet pair (SPP) flow control schemes are studied. Additionally, the alternating static window (ASW) flow control is defined and evaluated. Previously, it has been proven that the packet-pair scheme is stable for store-and-forward networks based on Rate Allocation Servers. The applicability of a PP flow control to wormhole-routing networks is studied and evaluated through simulation. It is shown that if high throughput is desired, ASW is the best method for controlling the average latency. On the other hand, if low throughput is acceptable, SPP can be applied to maintain low latencies.

1 Introduction

The term flow control refers to the techniques that enable a data source to match its transmission rate to the currently available service rate in the network and at the receiver [9, 11]. Apart from this main goal a flow control mechanism also should adhere to the following requirements: be simple to implement, use a minimum of network resources (bandwidth, buffers, etc.), and operate effectively when used by multiple sources. Additionally, the principles of fairness should be observed for shared resources. And finally, the entire networked system should be stable, i.e., for a constant configuration the transmission rate of each source should converge to an equilibrium value.

This paper considers two closed-loop flow control schemes - the static window and the packet pair flow control protocols. In the static window scheme [12] the source stops transmitting when in has sent a number of unacknowledged (outstanding) request equal to the size of the defined window. The main problem with this approach is that the optimal window size depends on many factors which vary over time and differ among connections. Therefore, choosing a single static window size that is suitable for all connections is impossible. In the packet pair scheme [8] the source estimates and predicts the network conditions based on the delay observed for a pair of consecutive packets and adjusts its transmission

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1148–1156, 2000.
© Springer-Verlag Berlin Heidelberg 2000

rate accordingly. The scheme is proved [8] to result in a stable system for store-and-forward networks based on Rate Allocation Servers.

This paper investigates the applicability of packet pair flow control to wormhole-routing networks that are not based on Rate Allocation Servers. Since the packet pair flow control does not limit the maximum number of outstanding requests, the static window protocol is employed in conjunction with it.

Flow Control in the ServerNet SAN.
The ServerNet system area network (SAN) is a wormhole-routed, packet-switched, point-to-point network with special attention paid to reducing latency and assuring reliability [4, 5]. It uses multiple high-speed, low-cost routers to rapidly switch data directly between multiple sources and destinations.

ServerNet implements two levels of flow control - hop-by-hop flow control and end-to-end flow control. Hop-by-hop flow control is performed by the exchange of special flow control flits (busy and ready) between the two devices connected through the link. Busy flits signal that the receiver queue is full. When the transmitting device receives a busy flit it ceases sending data until it receives a ready flit. End-to-end flow control is performed through the static window protocol. In this scheme each request packet has to be acknowledged by a response packet. The size of the static window limits the number of unacknowledged (outstanding) requests that can be transmitted. When a source reaches this limit it ceases transmitting requests until it receives at least one response.

Simulation Model.
The simulation model is discrete-event and unit-time [7]. Each device enters a particular state during each time step. The devices are activated in an random order. All performance measures collected during the course of the simulation are averaged over a number of packets sufficient to achieve the desired level of data accuracy for a confidence level of 95%. Collection begins when the system enters a steady state. Steady state is determined by the method of moving averages presented in [7]. The statistical data produced by the simulator was validated using experimental data collected at Compaq Tandem Labs. Discrepancies between the simulation and experimental results were found to be less that 5%. Since the simulation operates at a data accuracy of 3% these discrepancy are insignificant.

2 Packet Pair Flow Control

The packet pair flow control [9] (PP) belongs to the class of rate-based flow control protocols. PP estimates the conditions in the network by observing the time interval between the receptions of the responses to a pair requests of requests (packet pair) transmitted back-to-back. Moreover, it predicts the future service rate in the network and adjusts incorrect past predictions. The PP flow control is subject to the following limitations: packets must always be transmitted in pairs; the service rate of non-bottleneck servers is assumed to be deterministic.

To circumvent these limitations the simplified packet pair (SPP) flow control defined in [6] is described below.

Implementation of SPP.
The simplified packet pair flow control (SPP) is implemented as follows:

1. The inter-request delay is determined by a variable, I, which is 0 initially. After a packet is transmitted the next packet may not be transmitted before a time period of I expires.
2. The difference between the RTTs of every pair of consecutive packets to/from the same destination is compared with a threshold parameter *delta*. If *delta* is greated a "win" is registered.
3. A history of the last h comparisons is kept.
4. If the number of wins, h_W, are more than $\frac{h}{2}$, I is decremented by a value that depends on h_W. The greater h_W, the greater the decrement.
5. If $h_W < \frac{h}{2}$, I is incremented by a certain value that depends on h_W. The smaller h_W, the greater the increment.

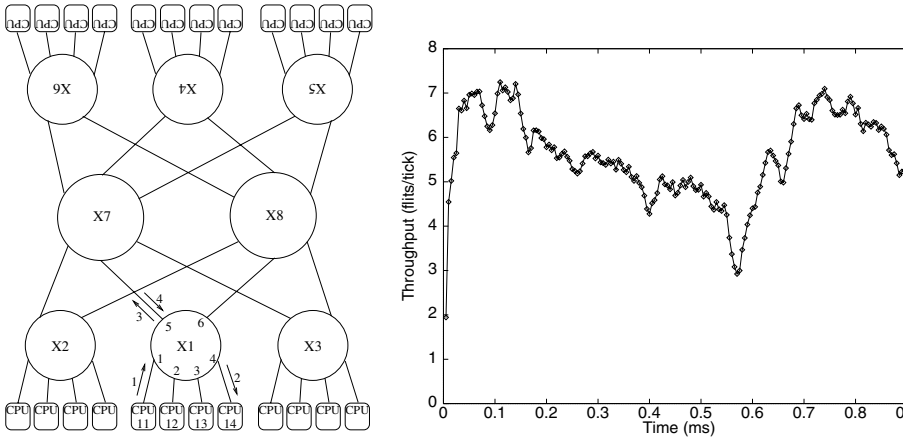
Evaluation of SPP.
Some statistics for the operation of SPP are shown in Table 1. They are based on the topology shown in Fig. 1-a with a Uniform traffic distribution and a generation rate of 200 requests/μs, which is selected to be past the saturation point of the network. Consider the statistics for the number of "wins" (Table 1-a). A window of $h = 8$ comparisons is kept. When the number of "wins" is equal to 4, SPP does not modify the inter-request delay. It can be concluded based on the average number of "wins" and the low deviation that generation rate controlled by SPP converges to an equilibrium state, i.e., the system is stable.

Average: 4.34	Average: 25.96
Std. Dev.: 1.16	Std. Dev.: 29.47
Minimum: 0	Minimum: 0
Maximum: 8	Maximum: 141
(a) Number of Wins	(b) Inter-Request Delay (μs)

Table 1. Statistics for SPP

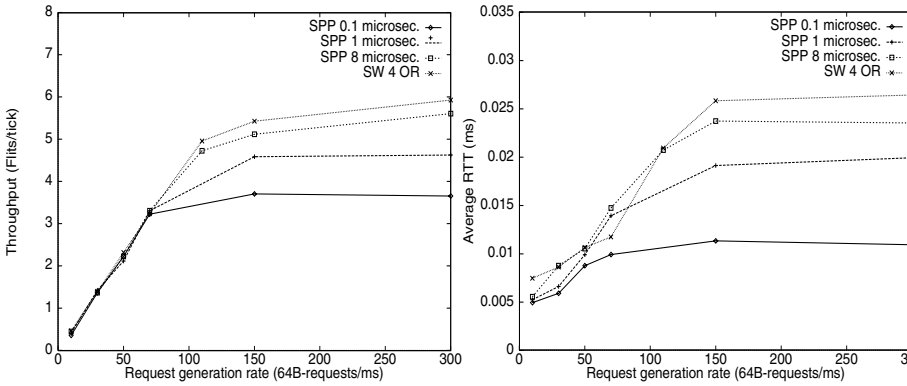
The inter-request delay statistics (Table 1-b) show that the SPP algorithm introduces a significant inter-request delay - an average of approximately 26 μs. During this period requests are held at the source devices. By adding the RTT (5.94 μs is the average observed in this case), the average request-to-response time totals approximately 32 μs. This value is higher than the RTT in the absence of the SPP algorithm, where the request-to-response time is equivalent to the RTT of 30.2 μs. It becomes apparent that the RTT is reduced in SPP by introducing an approximately equivalent delay at the source device. Essentially, the SPP scheme changes the location (source device vs. network and destination

device) where delays are incurred, but does not reduce the total request-to-response delay.



(a) 24-CPU Topology (b) Throughput snapshots

Fig. 1. Network Topology and Throughput Snapshots



(a) Throughput (b) Round Trip Time

Fig. 2. Performance of SPP, $delta = 0.1, ..., 8\mu s$

The effect of the parameter *delta* on the operation of the SPP algorithm in the 24-CPU topology shown in Fig. 1-a is evaluated for $delta = 0.1, 1, 8\mu s$. The results are presented in Fig. 2 and Table 2. The window size is limited to 4 outstanding requests (OR). The data shows that as *delta* is increased both the throughput and the RTT increase, growing closer to the performance characteristics achieved with the static window (SW) protocol alone. This trend is also observed in the average ORs, shown in Table 2.

It can be concluded that the parameter *delta* essentially limits the generation rate of devices. As *delta* is decreased, SPP introduces higher inter-request delays, thus reducing the generation rate. For low values of *delta*, a less requests are transmitted into the network. This results in low congestion and hence low RTTs. However, the low generation rate also leads to low throughput. Conversely, increasing *delta* leads to increases in both the RTT and the throughput.

	SPP *delta* (μs)			SW	ASW
	0.1	1.0	8.0		
Throughput ($flits/tick$)	3.6	4.63	5.61	5.93	5.53
Avg. RTT (ms)	10.9	19.9	23.9	26.4	21.9
Avg. OR (packets)	1.07	1.21	3.38	3.5	2.83

Table 2. Flow Control Schemes: Throughput, Average Round Trip Time (RTT), and Average Outstanding Requests (OR)

3 Alternating Static Window Flow Control

Ideally, the flow control scheme should maintain a high number of ORs to maximize throughput by overlapping (pipe-lining) the request-propagation and the request-processing delays but at the same time it should limit the number of ORs to minimize queueing delays at the end devices. In the SW scheme the number of ORs is maintained at maximum regardless of the delays, which leads to high throughput and high delays. An alternative approach is to halt the generation of requests when the high window mark is reached and to resume generation when the low window mark is reached. Reaching the high window mark is taken as an indication that the RTT is large, i.e., the network is overloaded. While the low window mark indicates that a sufficient amount of requests have been processed and it can be assumed that the network load has decreased to an acceptable level. It can be expected that this scheme will maintain high throughput because it pipe lines the requests, however it should lead to reduced queueing delays since high delays are detected and generation is halted until the network load is relieved.

To further support this conjecture we analyze the dynamic behavior of the network characteristics, based on the throughput and link usage snapshots shown in Figs. 1-b and 3. The link categories used in Fig. 3 are specified in Fig. 1-a. The following observations are made:

• initially (0.004 ms) there is no stalling of the links and transmission is not at 100%; this occurs because the transfer of data from memory to the interface has a start-up delay and the interface is not fully utilized;

• next (0.006 ms) there is more data available than the capacity of link 4 and stalling is observed;

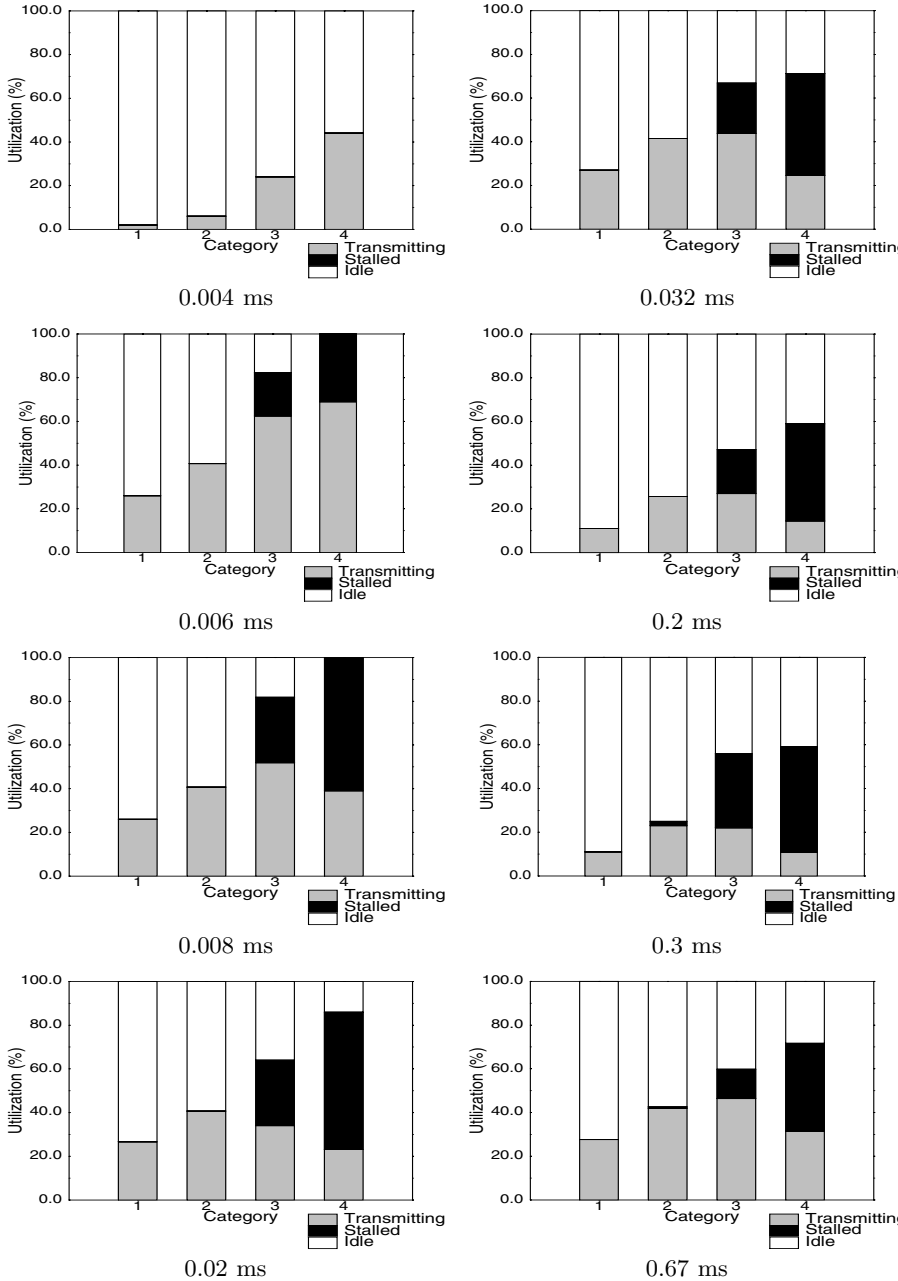
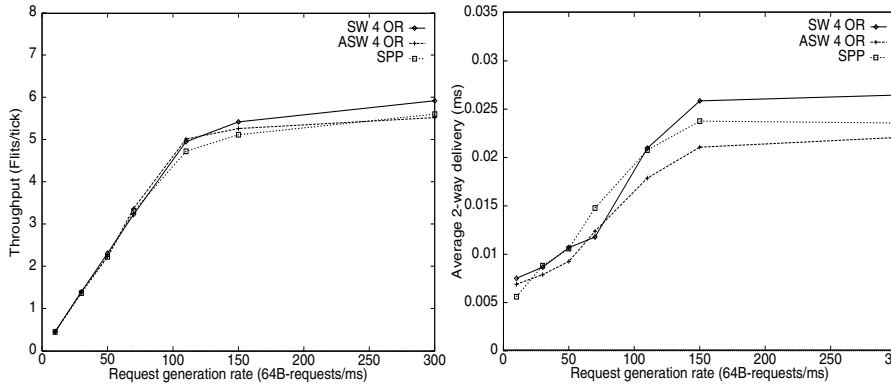


Fig. 3. Link Usage Snapshots for Static Window Flow Control



(a) Throughput (b) Round Trip Time

Fig. 4. Performance of Flow Controls: (1) ASW, 4-0 OR; (2) SW, 4 OR; (3) SPP, $delta = 8\mu s$

• the stalled time continues to increase until the static window limit (SWL) is reached and request transmission is halted, this causes idle time to appear at 0.01 ms and increase in proportion from there on; the idle time does not appear due to lack of data since it is constantly available; the increasing stalled time causes increasing delivery times;
• the increasing delivery times cause the SWL to be reached more often and remain in effect for longer periods, thus causing increased idle periods, during which packet generation is halted.

It is concluded that continuous transmission after the static window limit (SWL) is reached leads to a prolonged deterioration of the throughput, due to the increasing latencies displayed in the stalled time of link usage statistics. As shown in Fig. 1-b this deterioration continues for a period of time after which improvement is observed until the next period of deterioration commences. These trends alternate in a cyclic manner. It is desirable to maintain a controlled amplitude for this cyclic behavior, so that the average latency has a lower variance. One way to achieve a controlled amplitude is to halt transmission when the SWL is reached and to resume when the low window mark is reached. This would allow the network to recover from the large load and to transport the next burst of data more efficiently, with a lower latency. Such a scheme is implemented using the following rules.

Definition 3.1 *Alternating Static Window (ASW) flow control.*

1. *Transmission is allowed while the number of ORs is less than the high window mark (HWM);*
2. *once HWM is reached transmission is not allowed until enough acknowledgements are received to reduce the window size to the low window mark (LWM).*

The performance of the network with ASW ($HWM = 4$, $LWM = 0$), is presented in Fig. 4 along with that of the 4-OR SW and the SPP algorithm with 4 OR and $delta = 8\mu s$. It can be seen that all three approaches achieve approximately equivalent throughput. However, ASW displays an average RTT approximately 25% lower that the other two approaches. This demonstrates that if high throughput is desired, ASW is the best method for controlling the average latency. On the other hand, if low throughput is acceptable, SPP can be used to provide very low latencies.

4 Summary

The simplified packet-pair (SPP) flow control is evaluated. It is shown that the operation SPP can be adjusted by varying the value of the threshold parameter *delta*. The alternating static window (ASW), is defined. It is demonstrated that ASW achieves a throughput equivalent to that of SW and SPP with a large *delta*. Additionally, ASW displays a significantly lower (approx. 25%) average two-way delivery time. It is concluded that SPP is a flexible mechanism which allows sources to maintain different generations rates for different destinations. The performance of a system with SPP can be adjusted ranging from high throughput and high latency to low throughput and low latency. On the other hand, ASW is a significantly simpler mechanism that provides high throughput and reduced latency in comparison with SW and SPP. Consequently, if high throughput is desired, ASW is the best method for controlling the average latency. On the other hand, if low throughput is acceptable, SPP can be used to provide extremely low latencies.

References

[1] D. Avresky, V. Shurbanov, and R. Horst. The effect of the router arbitration policy on the scalability of ServerNet™ topologies. *J. of Microprocessors and Microsystems*, 21:545–561, 1997. Elsevier Science, The Netherlands.

[2] D. Avresky, V. Shurbanov, and R. Horst. Optimizing router arbitration in point-to-point networks. *J. of Comp. Comm.*, 22(5), April 1999. Elsevier Science, The Netherlands.

[3] D. Avresky, V. Shurbanov, R. Horst, W. Watson, L. Young, and D. Jewett. Performance modeling of ServerNet™ topologies. *The J. of Supercomputing*, 14(1), August 1999. Kluwer Acad. Pub.

[4] W. Baker, R. Horst, D. Sonnier, and W. Watson. A flexible ServerNet-based fault-tolerant architecture. In *Proc. of the 25th Int. Symp. Fault-Tolerant Computing*, pages 2–11, Pasadena, CA, June 1995.

[5] R. Horst. TNet: A reliable system area network. *IEEE Micro*, pages 37–45, Feb. 1995.

[6] R. Horst and P. Mehra. ServerNet Rate Control. Tandem Labs Technical Memorandum TL.17.2, Dec. 1998.

[7] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[8] S. Keshav. A control-theoretic approach to flow control. In *Proc. of SIGCOMM'91 Conf.*, volume 21 of *ACM Comp. Comm. Review*, pages 3–15, Zurich, Switzerland, September 1991.

[9] S. Keshav. *An Engineering Approach to Computer Networks.* Addison Wesley Longman, Inc., 1997.

[10] J. Kim and D. Lilja. A network status predictor to support dynamic scheduling in network-based computing systems. In *Proc. IEEE 13th Int. Par. Proc. Symp.*, pages 372–378, San Juan, PR, April 1999.

[11] S. Low and D. Lapsley. Optimization flow control - i: Basic algorithm and convergence. *IEEE Trans. on Networking*, 7(6):861–874, December 1999.

[12] C. Petitpierre and A. Zea. Implementing protocols with synchronous objects. In D. Avresky, editor, *Dependable Network Computing*, chapter 6, pages 109–140. Kluwer Acad. Pub., November 1999.

The WMPI Library Evolution: Experience with MPI Development for Windows Environments[1]

Hernâni Pedroso and João Gabriel Silva

CISUC
Universidade de Coimbra – Polo II
3030-397 Coimbra
Portugal
{hernani,jgabriel}@dei.uc.pt

Abstract. The usage of Windows based machines as a platform for parallel computing is rapidly increasing, mostly due to their excellent ratio cost/performance. The WMPI (Windows Message Passing Interface) was the first implementation of the MPI standard for Windows based machines. Originally based on the MPICH implementation, the library suffered several changes during the past years. This paper describes the evolution of the library since its first version. Recent changes have been introduced, which enable the dynamic creation of processes and the usage of simultaneous devices. This paper also describes the trends in the field that drove its evolution and the experience gathered through the development of the MPI standard for Windows clusters.

1 Introduction

The performance of Personal Computers has considerably increased in the past years. Their level of performance rivals with the much more expensive Unix workstations [1], especially in integer computation. Most of the parallel computing users cannot afford an MPP machine or a large number of workstations, hence the option of using PCs became common. A cluster of PCs today is considered a high-performance platform with a low cost/performance ratio. The number of PCs available in the institutions is also a source of a considerable computational power that may be used for commodity supercomputing.

WMPI (Windows Message Passing Interface) [2,3] was the first full implementation of the MPI standard [4] for Windows operating systems. MPICH [5] was the base of the first version, released in April 1996. WMPI used a Win32 port of the p4 [6] library to setup the environment and manage the communication between processes. By using the MPICH architecture it was also possible to maintain the compatibility between WMPI and the MPICH libraries in mixed clusters of Windows and Unix machines.

[1] This work was partially supported by the Portuguese Ministry of Science and Technology and the European Commission through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95 - ParQuantum.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1157-1164, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Although the base architecture was never changed, we conducted several modifications during the WMPI lifetime to improve performance and usability. A recent study, which evaluated implementations of MPI for Windows NT environment [7], considered WMPI as the best freely available implementation. In addition, the study concluded that WMPI rivals with other commercial implementations in performance and functionality. However, the release of the MPI-2 standard [8] and requirements for more functionality led to the development of a new internal architecture for WMPI. A completely new library, WMPI version 1.5, was recently released. This new architecture is the base for WMPI 2.0, an MPI-2 compliant version of the WMPI library, which will be release in the near future.

The MPICH architecture, which is the base for most of the existing libraries is presented. A section is dedicated to the evolution of the usage of PCs for high performance computing. The two WMPI architectures are presented as well as the major factors that determined the evolution of the library. Some lessons about Windows usage for MPI computing and implementation decisions are also presented.

2 Related Work

Several other implementations of MPI for Windows machines are available. There is a Windows version of MPICH developed by the Argonne National Laboratory (MPICH.NT) [9]. It uses shared memory for inter-process communication in the same machine and TCP/IP for remote processes. This implementation is still in its early steps.

MP-MPICH [10] is a multi-platform MPI implementation based on the MPICH architecture developed by RTWH Aachen. This implementation is available for Unix and Windows systems. It supports SCI [11], TCP/IP and shared memory communication mediums. The low latency in the communication that this implementation provides [12] is achieved by using active wait in the devices.

FM-MPI [13] is a MPI library built on top of the Fast Messages library [14]. It is based on the MPICH code. Relying on the Fast Messages implementation, FM-MPI has support for Myrinet and TCP/IP.

All the above libraries are freely available, however there are also some commercial implementations of MPI for Windows environments. MPI/Pro [15], from MPI Software Technology, Inc and PaTENT MPI [16] commercialized by Genias Software GmbH. MPI/Pro is available for VIA, TCP/IP and shared memory intercommunication mediums. PaTENT MPI is provided with TCP/IP and shared memory support, and is actually an evolution of WMPI.

3 MPICH – The WMPI's Base Architecture

The development of the MPICH library occurred along with the elaboration of the MPI standard. The objective of the Argonne National Laboratory/Missisipi State University development teams was to provide early feedback to the MPI Forum by creating a test bed to evaluate the correctness of the decisions. The library also aimed

to enable the high performance community to experiment and use a standard implementation of MPI as early as possible.

The aims of the architecture design were portability and efficiency. By freely providing the library, as well as the source code, the development team allowed the parallel computing community to rapidly use a reliable library that implemented the whole MPI interface (version 1 of the standard). Due to its portable characteristics, both the original development team other research institutions and software/hardware vendors developed several versions of the library for many different systems.

4 Windows Clusters Environment

In 1996, when the first version of WMPI was released, dedicated clusters of PCs were practically unknown. The base idea of the library was to use the available PCs in the institutions that were being used for small interactive tasks (e.g. text editing and e-mail reading). The PC's owner wishes to keep the response time of the computer to its interactive actions. Hence, it was important that WMPI was as little intrusive as possible. For example, the use of polling was completely unacceptable because it would significantly slowdown the other processes, running on the same CPU, even when no calculation was occurring.

A heterogeneous network of PCs and Unix workstations was very common. In the beginning, PCs were used to occasionally boost the computational power of the Unix workstations. PCs, due to their continued increase of performance, became more common for parallel computing. The use of common off-the-shelf components for creating clusters with a considerable computational power became normal. The computational power of each unit of the cluster had a considerable boost with the appearing of SMPs with two or four processors. The decision of purchasing PCs for high performance computing became normal, since it was a low investment with practically null risk.

One of the drawbacks of constructing a PC cluster for high performance computing was the lack of interconnection networks that could present high bandwidth and low latency between nodes. Most of the PC clusters were using TCP over Ethernet and Fast Ethernet networks, which are not optimized for message exchange performance but to reliability and cost. As the computational power of the PCs grew, the network became the bottleneck of the cluster. Aware of this fact, the hardware vendors have started to create new technologies that improve the message passing performance between the computer nodes. VIA is the most recent effort, although SCI, Myrinet and Gigabit Ethernet are also available.

5 The First WMPI Architecture

When releasing the first MPI library for parallel computing using Windows PCs, it was very important to allow it to cooperate with a MPI library running on Unix workstations as well. Although the idea of using PCs for high-performance computing was unusual, the possibility of using a heterogeneous environment seemed interesting

and promising for most of the parallel computing community. Hence, it was important that WMPI could cooperate with MPICH running on Unix workstations.

WMPI was based on the existing MPICH implementation. For the sake of compatibility with Unix workstations, p4 was chosen as the communication subsystem that runs under the MPICH's ADI. This decision was also the one that required a smaller development time, since p4 was very stable and a common communication subsystem for the MPICH library. Due to the excellent work of the MPICH developers, the upper layer was easy to port. P4 was the main concern, because it directly interacts with the operating system. P4 handles with two types of communication mediums: shared memory and TCP/IP. Processes running on the same machine use shared memory, while TCP/IP was used for exchanging information with remote processes.

WMPI avoided any form of active wait. Any thread that needs to wait for some event to occur (typically waiting for a message) stops competing for the CPU and does not use its entire quantum. This was very important considering the environment that WMPI was addressing.

During its lifetime, several problems were solved in the library. The development experience enabled to improve its performance. This resulted in a very stable and mature MPI library. Several thousands of users and institutions have already downloaded WMPI from our web site [3] and are actively using it.

6 Multiple Devices

New technologies have emerged to improve the message passing performance in clusters. VIA, SCI, Myrinet and Gigabit Ethernet are good examples of the recent efforts. The MPI library must be able to follow these improvements on the underlying systems. It is thus necessary to create specific devices for each technology in order to use the maximum performance that it can offer. The necessary resources to produce an efficient device for a new intercommunication media are considerable. It was urgent to create an architecture that enabled a rapid development of new WMPI devices for different communication media. In addition, it should be possible to create a device without knowledge of the library internals. This would enable third-party institutions (e.g. hardware vendors) to create devices for their technology, hence providing WMPI in a wide range of communication mediums.

It is common to find more than one type of technology for communication in a cluster. The most common configuration is, probably, shared memory and TCP/IP. However, with the arrival of new technologies and with the falling of prices, other configurations can be found. Moreover, when upgrading an existing cluster, the new nodes can use a different communication medium than the older ones. The older nodes can be used in conjunction with the newer computers to increase the performance of the cluster to solve some important problem. Institutions may also desire to connect separated clusters, which might use different communication media. All these possibilities create a wide range of possible configurations for a MPI execution. The possibility of easily configuring the computation to use any number of specific devices, according to the cluster configuration, is presently considered very important.

7 Dynamic Environment

MPI was well accepted by the community and became a *de facto* standard for parallel computing. Eager for more functionality, both users and developers presented several requests to the MPI Forum to extend the MPI standard. Version 2.0 of the standard (MPI-2) includes significant new capabilities, from which dynamic process creation is a good example.

The ability of using a dynamic process environment within MPI-2 improves the usability of the library for several types of applications. An application can decide in runtime the number of processes that are required to manage the data to be processed. Processes can be added or subtracted to the computation according to its needs. It can also adjust the number of processes according to the level of parallelism of each phase of the computation. Processing units or entire clusters can join at any time a client/server application, since MPI-2 allows for the joining of separately launched MPI computations.

But the new chapters of MPI-2 require deep changes in existing MPI libraries. For instance, with the introduction of dynamic creation of processes, it is possible for processes not belong to the `MPI_COMM_WORLD`, that in MPI-1 was a global communicator. These processes are the result of a process spawn (creation) or the joining of two MPI computations. When a spawn is performed or two MPI computations join not all the processes of both computations have to be involved. This means that each process will have its own set of processes with which it can communicate. Although when the new processes join, the communication has to be performed through an inter-communicator, they can form an intra-communicator using the `MPI_Intercomm_merge` function. It is also possible to extract the remote group of processes in an inter-communicator and manipulate it as any other group. This implies that an inter-communicator can be closed (which should indicate that the two computations are independent) but some of the processes still communicate, through groups and intra-communicators that were built using the disconnected inter-communicator. This situation is an example of the extreme freedom that the users have to manipulate the processes and create the configuration of interconnections and communicators. The usage of ranks in `MPI_COMM_WORLD` to globally identify the processes is impracticable in such environment. It is necessary to create another form of global identification, which must be valid whether the process starts within the same computation or joins in runtime.

8 The Second WMPI Architecture

The new MPI-2 features lead us to design a completely new architecture for WMPI, to avoid having to do successive patches, which are more prone to errors and render the evolution difficult. Based on our experience with cluster computing and considering the trends on the field, several other reasons were identified that strongly required a new architecture. The design aimed to create an architecture that is able to:

- manage a dynamic environment, where processes could be created and destroyed, join and leave the MPI computation;

- work with several devices simultaneously;
- easily support new communication technologies by reducing the complexity, hence the development time, of the communication dependent code;
- be completely thread safe;
- diminish the communication latency;
- increase communication throughput.

By creating our own architecture, the compatibility to the MPICH library is lost, but WMPI gains the freedom to pursue its own goals without relying on the others pace. The new architecture does not try to be portable across other platforms. Since the WMPI library is used in Windows environments, we decided to use the features that the operating system provides. This reduces the complexity and increases the performance of the library.

Thread safeness and performance were a constant concern during the design and implementation of the library's architecture. Every structure and all the functions that manipulate its data were studied to verify if they have or not to be thread safe. The synchronization points were reduced to the minimum to improve the general performance of the library.

The user is responsible for defining how processes communicate during the WMPI computation through a cluster configuration file. Devices are now independent DLLs that are loaded at the startup of each process according to the cluster configuration. WMPI has the capability to interact with any number of different devices simultaneously. Within each process, WMPI associates a device to each machine of the cluster, according to the cluster configuration. When one MPI process needs to interact with another process, it chooses the correct device and performs the necessary action.

During the design of the new architecture, we identified the necessary operations that the devices should perform. It was important to reduce the expected functionality of the devices to a minimum, since it should be feasible to produce a device for every possible technology. Moreover, simple devices are easier to optimize and guarantee the independence of the library core.

Each process contains a structure to represent each process with which it is connected. This structure has one record per process, containing the set of addresses of the process that it represents. Each record also contains a reference to the device that must be used to communicate with the process. This set of records represents all the processes in the computation with which the process that owns the structure can communicate, because it has at least one group in common with each of them.

This new structure is already implemented and available in the WMPI 1.5 version, which can be downloaded from the WMPI web site [3].

9 Lessons Learned

During the development and design of the WMPI library, we came to some conclusions about cluster computing, implementation and design decisions. This section describes some of the most important ones:

- **Windows Clusters**: Dedicated clusters are still a small minority in the universe of Windows PCs used for parallel computing. WMPI is commonly used in clusters shared by several users, without any major management of the resources. Moreover, many of the applications run in a set of computers that are also used for interactive tasks. Hence, it is very important to make the MPI processes use fairly the available resources, since they are shared with other processes running in the same machine (MPI processes or others). In this environments the usage of polling is self-defeating, since the processes start to stall each other and the whole systems runs slower.
- **Resources Access**: In a Windows environment, resources are shared among all the processes in the system. Moreover, each process may have more than one thread. These threads compete to access to the system resources. The operating system automatically manages the access to this resources. We have verified that in most of the occasions is more effective that the processes control the access and allow, through synchronization, only one thread to use the resource at a time. The results were more perceptible in SMPs, where more synchronization is necessary from the operating system.
- **Windows Synchronization API**: The Windows API provides several synchronization functions. Through exhaustive tests, we concluded that considerable latency differences exist between them. It is thus important to carefully choose the one that best fits each synchronization requirements.
- **Synchronization Methods**: The synchronization is responsible for most of the latency in the communication. The penalty of having a wrong synchronization design is enormous. The WMPI 1.5 version suffered several changes in the architecture to diminish the communication latency and an improvement of 50% was achieved when compared with the first tests.

10 Conclusions

Due to the increase of computational power of PCs, their use became common for high-performance computing. Clusters of PCs, due to their low price, allowed small companies and universities to have access to a parallel computing platform to speedup their applications. The intercommunication technology became the bottleneck of the clusters, since the computational power of the PCs increases rapidly. New intercommunication technologies have emerged in the past years and are now becoming common as their price gets lower. Today it is possible to build a cluster with an enormous performance at a quite low price.

WMPI was the first MPI implementation for Windows based machines, its first release having been, in 1996. Originally based on the MPICH implementation, it evolved through the years and became a very stable MPI implementation with a good performance. The number of users has increased along with the usage of PCs for high performance computing.

The release of the MPI-2 standard brought new features that, together with other important demands, like thread safeness and the ability to work with multiple devices simultaneously, required a complete change of the internal architecture of the library, leaving behind the MPICH architecture that proved not be capable of withstanding

those changes. This new architecture will, hopefully, form the basis of a high performance stable MPI-2 library for Windows based clusters.

References

1. CPU Info Center – http://bwrc.eecs.berkeley.edu/CIC/.
2. Marinho, J. and Silva, J.G.: WMPI – Message Passing Interface for Win32 Clusters. Proc. of 5th European PVM/MPI User's Group Meeting, pp.113-120 (September 1998).
3. WMPI Homepage – http://dsg.dei.uc.pt/wmpi
4. Message Passing Interface Forum: MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4):165-414 (1994).
5. Gropp, W., Lusk, E., Doss, N. and Skejellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing Vol. 22, No. 6, (September 1996).
6. Butler, R. and Lusk, E.: Monitors, messages and clusters: The p4 parallel programming system. Parallel Computing, 20:547-564 (April 1994).
7. Baker, M: MPI on NT: The Current Status and Performance of the Available Environments. NHSE Review, Volume 4, No 1 (September 1999).
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. (June 1997), available at http://www.mpi-forum.org.
9. MPICH for WindowsNT Download Page - http://www-unix.mcs.anl.gov/~ashton/mpich.nt.html.
10. MP-MPICH: Multiple Platform MPICH - http://www.lfbs.rwth-aachen.de/~joachim/MP-MPICH.html.
11. Hellwagner, H., Reinefeld, A.: SCI: Scalable Coherent Interface – Architecture and Software for High-Performance Compute Clusters. Lecture Notes in Computer Science Vol. 1734, Springer, ISBN 3-540-66696-6 (1999).
12. Performance of NT-MPICH - http://www.lfbs.rwth-aachen.de/~karsten/projects/nt-mpich/performance.html.
13. Lauria, M., Chien, A.: MPI-FM: High Performance MPI on Workstation Clusters. Journal of Parallel and Distributed Computing, Vol. 40, No. 1, pp. 4-18 (January 1997).
14. Pakin, S., Karamcheti, V. and Chien, A.: Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. IEEE Concurrency, vol. 5, no. 2, April-June 1997, pp. 60-73 (1997).
15. MPI Software Technology, Inc – http://www.mpi-softtech.com.
16. Genias Software GmbH, PaTENT – Parallel Tools Environment on NT, http://www.genias.de/products/patent/index.html.
17. The Beowulf Project – http://www.beowulf.org.

Implementing Explicit and Implicit Coscheduling in a PVM Environment*

Francesc Solsona[1], Francesc Giné[1], Porfidio Hernández[2], and Emilio Luque[2]

[1] Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain.
{francesc,sisco}@eup.udl.es
[2] Departamento de Informática, Universitat Autònoma de Barcelona, Spain.
{p.hernandez,e.luque}@cc.uab.es

Abstract. Our efforts are directed towards the understanding of the coscheduling mechanism in a NOW system when a parallel job is executed with local workloads, balancing parallel efficiency against the local interactive response. Explicit and implicit coscheduling techniques in a PVM-Linux NOW (or cluster) has been implemented. Their performance and overheads executing local tasks and representative distributed benchmarks have been analyzed and compared.

1 Introduction

Over the years, researchers have been developing time-shared distributed schedulers using coscheduling techniques, trying to adapt them to the new situation of mixing local and parallel workloads [1], [2], [3], [4] and [5].

Explicit coscheduling, all processes in a parallel application are scheduled simultaneously, with coordinated time-slicing between them. Generally, this yields good parallel program performance and this is widely used to schedule parallel processes involving frequent communication [1]. Coscheduling will ensure that no process will wait for a non-scheduled process for synchronization/communication and will minimize the waiting time at the synchronization points.

Two-phase spin-block synchronization primitives used for *dynamic coscheduling*, named *implicit coscheduling* in [2], [3] and [4], only requires processes to block awaiting messages arrivals for coscheduling to happen. With two-phase spin-blocking, the waiting process spins for a determined time and if the response is received before the time expires then continues executing else the requesting process blocks and another one is scheduled.

Algorithms for implementing new explicit and implicit coscheduling environments are presented in this paper. Extensive performance analysis, as well as studies of the parameters and overheads involved in the implementation, demonstrated the applicability of the proposed algorithms in these new environments.

* This work was supported by the CICYT under contract TIC98-0433

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1165–1170, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2 Coscheduling

In this section, the methods and metrics to measure their cost for explicit and implicit coscheduling distributed tasks in a PVM-Linux NOW are described.

2.1 Explicit Coscheduling

The aim of explicit coscheduling is to schedule all the distributed tasks in the cluster at the same time and let them execute during a period of time. From one global controller process running in one node named *master*, control messages are sent (in a broadcast form) to every explicit process (named *dts*) running in the composing workstations of the cluster, which are responsible for implementing explicit coscheduling. One of these control messages (*init*) informs all the *dts* processes to start delivering STOP and CONTINUE signals to their local high-priority distributed processes at regular intervals (see also [5]). The time spent in starting (T_{start}) all the distributed tasks is:

$$T_{start} = W_s(local) + W_w(dts) + S_{sig}(CONT) + W_w(dis) + W_s(dts), \quad (1)$$

where W_w/W_s is the elapsed time in waking up/suspending *dts*, a local task (*local*) or a distributed task (*dis*). $S_{sig}(CONT)$ is the maximum elapsed time in sending a CONTINUE signal to all the distributed tasks in the node. The time spent in stopping (T_{stop}) all the distributed tasks is:

$$T_{stop} = W_s(dis) + W_w(dts) + S_{sig}(STOP) + W_w(local) + W_s(dts), \quad (2)$$

where $S_{sig}(STOP)$ is the maximum elapsed time in sending a STOP signal to all the distributed tasks in the node. Because the time in delivering a signal to a group of processes does not depend on the signal to deliver, we consider that $S_{sig}(STOP) = S_{sig}(CONT) = S_{sig}$. Similarly, the values $W_w = W_s = W$ are considered to be equal. In consequence 1 and 2 can be reformulated as:

$$T_{ex} = T_{start} = T_{stop} = 4W + S_{sig}. \quad (3)$$

2.2 Implicit Coscheduling

The implicit coscheduling aim is to schedule only communicating distributed tasks at the same time. We are interested in only spinning the tasks during at most a context-switch period and not in spinning during the deliver of a round-trip message as in [2,3,4], as distributed tasks can follow many types of communication patterns and the messages can arrive asynchronously to distributed tasks, at any time. The metric T_{im} is used to compute the maximum overhead added in spinning, which also gives us a first reference to choose the *spin interval* (*sp*):

$$T_{im} = W_s(dis) + W_w(local) \quad (4)$$

Algorithm 1 *ImCoscheduling.* Implements the *implicit coscheduling.*

```
Initialize input_time, execution_time, sp
while (no_new_fragment) and (execution_time ¡ sp)
  and (execution_time ¡ timeout) do
      execution_time = current_time - input_time
if (no_new_fragment)
    if (timeout) then block (timeout - execution_time)
        else block (indefinitely)
```

Algorithm 2 *OneFragment.* Reads the fragment in only one phase.

```
call pvm_receive and wait until the fragment arrives
read the whole fragment ( header + body )
```

3 Algorithms

The implemented algorithms detailed in this section show how new distributed environments were created.

Function *ImCoscheduling* (Algorithm 1) implements implicit coscheduling by realizing a spin-block while the fragments (unit of PVM transmissions) composing a message are read. Algorithm 2, called *OneFragment,* reads each fragment in only one phase, instead of twice, as PVM does. Both algorithms were implemented in the *pvm_recv*( ) PVM routine.

Algorithm 3, *called Priority* was implemented outside the PVM, in a process named *Priority*, a copy of which is in each node of the cluster. It is responsible for assigning a high priority to distributed tasks. To do this only is necessary to assign a high priority (one unit level less than *Priority*) to *pvmd* in its creation.

4 Experimentation

The experimentation has been performed in a Now made up of an interconnection network of 100 Mbps Fast Ethernet and four PVM-Linux PCs with the same characteristics: 350Mhz Pentium II processor, 128 MB of RAM, 512 KB of cache.

A distributed application, *sintree* was implemented to measure performance of the implemented environments. It attends for a communication pattern of one to vary, and vary to one. *sintree* accepts two arguments: number of processes (M) and number of iterations (N). By default $M = 4$ and $N = 30.000$. Also, two kernel benchmarks (class A) from the NAS parallel benchmarks suite [6] were used: *is* and *mg*. In all the benchmarks, the communications between remote tasks was done through *RouteDirect* PVM mode.

4.1 Implemented Environments

The next distributed environments were created. The algorithm(s) used to implement each model is in parenthesis. PVM: original PVM. SPIN (1): the spin-block

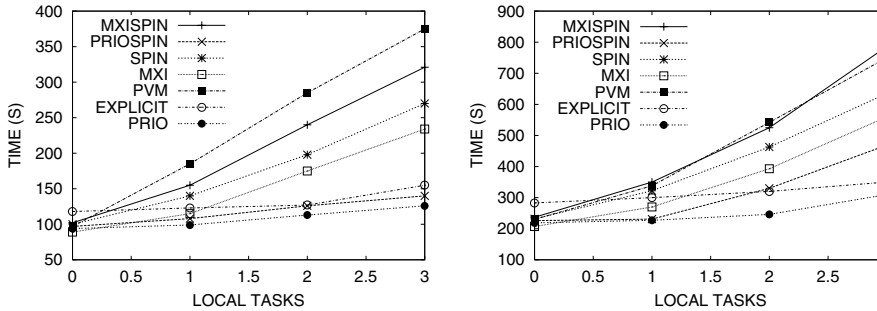


Fig. 1. *sintree* execution. (left) N = 30000. (right) N = 70000.

is only performed in the reading of the data fragment. MXI (2). MXISPIN (1, 2). PRIO (3). PRIOSPIN (1,3). EXPLICIT: periodically, after 90000 μs the *dts* daemon in each node delivers a STOP signal to all the local distributed processes and then, elapsed 10000 μs, *dts* delivers a CONTINUE signal to reawaken them. The measured $T_{im} \simeq 10$ μs, so in the spin models an *sp* of 10 μs was chosen.

Algorithm 3 *Priority.* Assigns a high priority to distributed tasks.

fork&exec (*pvmd*)
set priority (*pvmd* = max_priority - 1)

4.2 Results

Distributed Tasks Performance Fig. 1 shows the *sintree* execution times executing in the seven above cited modes while the local workload in each node (simulated by compiling applications) is varied from 0 to 3.

As was expected, optimal execution of the PRIO case can be observed. EXPLICIT without tasks is the worst mode, by increasing the workload, its performance scarcely decreases due to T_{ex} does not vary. MXI and SPIN modes scale fine and their performance is always between the PVM and PRIO. SPIN is faster than PVM because avoids a lot of times the blocking overhead in receiving messages. The PRIOSPIN case gives worse results than PRIO, as the unnecessary spin-block phase added in the first mode, this only adds an unnecessary overhead in the reading of the fragment. MXISPIN works worse than MXI, as in this case penalties when time-slice expires are more than ones in context switching.

Fig. 2 shows the results obtained from executing *is* and *mg* in the different models. The behavior of *mg* is similar to the *sintree* one. On the other hand, *is* does not work as fine as *mg* and *sintree* in the SPIN cases.

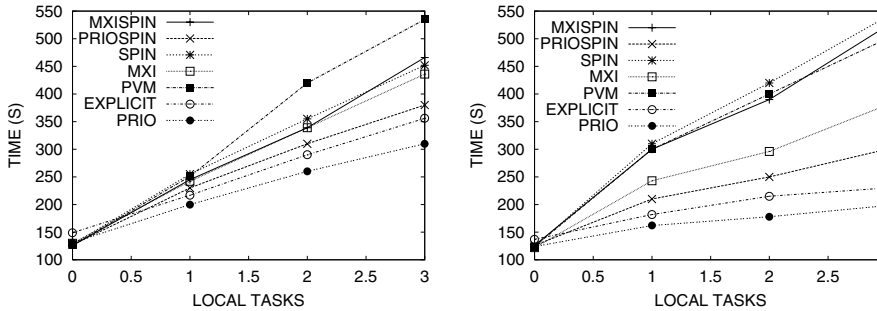


Fig. 2. Execution of the NAS parallel benchmarks (left) *mg* and (right) *is*.

Table 1. slowdown of a compiling local task.

slowdown	PRIO	PRIOSPIN	EXPLICIT	SPIN	MXI	MXISPIN
sintree	1.4	1.4	1.4	3.6	1.4	3.6
is	2.8	2.8	4.2	2.1	0	2.1
mg	90	92	42	8	1.6	8

Local Tasks Performance The influence of the models in the local tasks was based on measuring the slowdown calculated as follows:

$$sd_{MOD} = \frac{TMODEL - TPVM}{TPVM} 100,$$

where TMODEL (TPVM) is the execution time of a local task (a compiling application) when it was executed in such model (original PVM). See Table 1.

As might have been expected, when intensive message-passing distributed applications are executed (*sintree* and *is*), no effect in the local task is produced. On the other hand, a high slowdown is introduced if intensive CPU distributed tasks are executed (*mg*). As was to be expected, the explicit model has a great impact on the local task and even more in the PRIO and PRIOSPIN cases.

5 Conclusions and Future Work

In a PVM environment made up of a NOW of Linux nodes, we have implemented and discussed different coscheduling techniques and compared their performance. Also, we have discussed their main advantages and drawbacks. We are interested in developing a dynamical model and new coscheduling techniques for that environment.

References

1. Ousterhout, J.K.: Scheduling Techniques for Concurrent Systems. In Third International Conference on Distributed Computing Systems. (1982) 22–30.

2. Arpaci, R.H., Dusseau, A.C., Vahdat, A.M., Liu, L.T., Anderson, T.E., Patterson, D.A.: The interaction of Parallel and Sequential Workloads on a Network of Workstations. SIGMETRICS'95. (1995). 267–278.
3. Arpaci, R.H., Dusseau, A.C., Culler, D.E., Mainwaring, A.M.: Scheduling with Implicit Information in Distributed Systems. SIGMETRICS'98. (1998).
4. Dusseau, A.C., Arpaci, R. H., Culler, D. E.: Effective Distributed Scheduling of Parallel Workloads. SIGMETRICS'96 . (1996).
5. Solsona, F., Giné, F., Hernández, P., Luque, E.: Synchronization methods in distributed processing. IASTED AI'99. (1999) 471–473.
6. Bailey, D. et al.: The NAS parallel benchmarks. International Journal of Supercomputer Applications. vol. 5 no. 3 (1991) 63–73.

A Jini-Based Prototype Metacomputing Framework

Zoltan Juhasz[1,2] and Laszlo Kesmarki[1]

[1] University of Veszprem, Department of Information Systems, Veszprem, Hungary,
juhasz@irt.vein.hu

[2] University of Exeter, Department of Computer Science, United Kingdom

Abstract. This paper investigates the possible use of Jini technology for building Java-based metacomputing environments. The paper presents the structure of a prototype Jini metacomputing system. The overall working mechanism and some details of the implementation are described. It is shown that Jini is a suitable technology that deserves the interest of the metacomputing research community.

1 Introduction

The past few years generated tremendous interest in metacomputing. Several research groups world wide are working on how to provide seamless access to the vast number of computers connected by the Internet, to create better and more useful computing services, and to improve the performance of metacomputing systems. Different approaches, languages and paradigms have been used [3][4][5][6][7][8].

Jini technology [2] is a new Java-based technology to create autonomous, ad hoc networks of digital devices. It is a service-based system featuring automatic discovery of the network and its available services, fault tolerance, transactions and distributed event mechanism.

The purpose of this paper is to report on our ongoing work whose aim is to investigate the suitability of Jini as a metacomputing technology, and outline the structure and underlying working mechnism of one possible Jini metacomputing system.

2 Metacomputing Systems

The term *metacomputing system* [9] now commonly refers to systems composed of potentially thousands of personal computers, workstations or supercomputers to solve compute-intensive problems or provide better solutions in geographically distributed work environments.

Compared to parallel systems, metacomputing environments open up a set of new problems and issues that must be dealt with. Successful systems must cope with partial failure, geographical distance, large network latencies, different

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1171–1174, 2000.
© Springer-Verlag Berlin Heidelberg 2000

language environments, hardware platforms and operating sytems, and must provide solutions for security problems, fault tolerance and portability [1].

Most metacomputing systems are service based. Services provide distinct groups of functionality and are generally used for resource management (allocating/scheduling tasks, load-balancing, process management), communication (eg message passing), security and authentication, system status (monitoring the availability and system load of components), and accessing remote data. These are nicely represented by the service set of the Globus [3] project, such as the Resource management, Communication, Security, Information, Health and Status, Remote Data Access, and Executable Management services.

3 A Minimal Metacomputing System

Our current plan is to create minimal system (a testbed), which we can use to experiment with various features of Jini and consequently, evaluate its general suitability for building metacomputing systems. We expect the prototype system to be able to *(i)* monitor the current state of the computing resources, *(ii)* identify which resources it should use for parallel tasks and *(iii)* map and execute the tasks on the selected resources.

We use the following three participants, Client, Broker and Host. The *Client* represents the parallel application to be run on the system. A client first has to find available brokers then request the execution of the program from one or more. A *Broker* is responsible for two main actions. It must be able to *(i)* monitor resources and *(ii)* select a subset of them for executing the client's tasks. Monitoring is performed through the *Host* services that provide crucial state information and are able to execute tasks allocated to them by the broker service.

3.1 The Operation of the System

The system is depicted in Fig 1. Each processor runs a host service that implements a generic compute engine and contains methods that manage processes. The host service also contains static attributes describing the type of computer it represents. The service is lease based, and the attributes are updated on each lease renewal.

Brokers discover potential computing resources by finding Host services in the registrars and —after downloading the service— extract information from the host services. In the simplest form of implementation, the Host service will run entirely in the broker's virtual machine. Note that it is possible to create host services that provide dynamic, time-varying information (e.g. change in load level). In this case, the downloaded host service cannot run exclusively in the VM of the broker, as it has to obtain information from the originator host. Thus, in this case, the host service will only provide a proxy for the service that stays on the host.

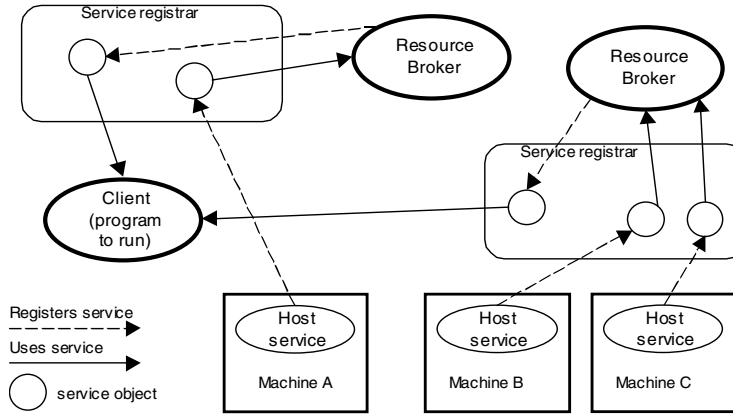


Fig. 1. Movement of service and application objects during the operation of the system

4 Implementation of the Prototype System

Each service registers with the registrar through the registrar object's `register()` method. This has two input parameters, a `ServiceItem` object describing the service and a lease period given in milliseconds. A `ServiceItem` object comprises a service ID, the actual service object representing the service, and an attribute set describing the service for clients during the template-matching based service lookup operation.

4.1 The Host Service

The Host service must implement a generic compute engine. In addition we store static system information such as host URL, the number and types of CPUs, the number and types of network interfaces of the host, network latency and bandwidth, topology, memory size and location.

The Host service implements a `public Object execute(Task t)` method to execute client task `t` of type `Task` interface. System and state information are represented as Jini attributes. We have abstracted out attributes such as Processor, Network and Memory for our ptototype.

4.2 The Broker Service

The responsibility of the Broker service is to take an execution request from the Client, find suitable machines for execution, then allocate and run tasks on these machines. Allocation is based on requirements received with the execution request (e.g. required performance, maximum number of tasks, maximum allowed cost of execution to be payed for, etc) as well as on machine information extracted from Host services. In our prototype the priorities in task scheduling is

to minimize the number of processors and maintain geographical locality of the selected machines. This can only be violated if such a parallel supercomputer is available that can handle the execution of the complete problem itself.

During execution, the broker first looks for available Host services, then registers with the lookup service to become available to clients. The main methods of the Broker are `findMachines()` and `execute()`. The method `findMachines()` takes a list of execution requirements from the Client and returns a set of machine list–cost pairs. If the set is not empty, the Client can invoke the `execute()` method either on specific hosts or on arbitrary ones selected by the broker by specifying allocation priorities such as `minimum_cost`, `maximum_speed`.

5 Conclusions

This paper presented early results of the Jini metacomputing project. It outlined the structure of a possible Jini service-based metacomputing framework and described some details of the Host and Broker service implementation.

Our short term plan is to complete the implementation of the framework and experiment with it. There are several issues that will be have to be addressed in the future. Scalability of the system requires that increasing the size of the system does not create bottlenecks in service access and use. We will explore possibilities of creating a network of registrars as well as an array of brokers to avoid the system relying on a single or small number of registrars and brokers. Automatic execution mode will require negotiation between client and broker and among brokers, therefore we will investigate the potential of using multi-agent techniques in coordinating program execution.

References

1. M. Baker and G. Fox, Metacomputing: Harnessing Informal Supercomputers in R. Buyya (ed.) High Performance Cluster Computing Vol. 1, Prentic Hall, 1999.
2. W. Keith Edwards, Core JINI, Prentice Hall 1999.
3. I. Foster and C. Kesselman, The Globus project: a status report,Future Generation Computer Systems 15 (1999) pp 607-621.
4. A. Grimshaw, W. Wulf et al. The Legion Vision of a Worldwide Virtual Computer. Communications of the ACM, vol. (40)1, January 1997.
5. T. Haupt, E. Akarsu, G. Fox and W. Furmanski, Web based metacomputing, Future Generation Computer Systems 15 (1999) pp 735-743.
6. M. Migliardi, V. Sunderam, The Harness Metacomputing Framework, in Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, S. Antonio (TX), USA, March 22-24, 1999.
7. M.O. Neary, B.O.Christiansen, P. Capello and K.E.Schauser, Javelin: Parallel Computing on the Internet, Future Generation Computer Systems 15 (1999) pp 659-674.
8. L.F.G Sarmenta and S. Hirano, Bayanihan: building and studying a web-based volunteer computing systems using Java, Future Generation Computer Systems 15 (1999) pp 675-686.
9. L. Smarr and C. E. Catlett, Metacomputing, Communications of the ACM, Vol. 35, No. 6, 1992, pp 44-52.

SKElib: Parallel Programming with Skeletons in C

Marco Danelutto and Massimiliano Stigliani

Department Computer Science – University of Pisa
Corso Italia, 40 – I-56125 Pisa – Italy
marcod@di.unipi.it, massimiliano.stigliani@ac.com

Abstract. We implemented a skeleton library allowing the C programmer to write parallel programs using skeleton abstractions to structure and exploit parallelism. The library exploits a SPMD execution model in order to achieve the correct, parallel execution of the skeleton programs (which are not SPMD) onto workstation cluster architectures. Plain TCP/IP sockets have been used as the process communication mechanism. Experimental results are discussed that demonstrate the effectiveness of our skeleton library.[1]
Keywords: skeletons, parallel libraries, SPMD.

1 Introduction

Recent works demonstrated that efficient parallel applications can be easily and rapidly developed by exploiting skeleton based parallel programming models [5, 16, 15, 3]. With these programming models, the programmer of a parallel application is required to expose the parallel structure of the application by using a proper nesting of skeletons. Such skeletons are nothing but a known, efficient way of exploiting particular parallel computation patterns via language constructs or library calls [9]. The skeleton implementation provided by the support software takes care of all the implementation details involved in parallelism exploitation (e.g. parallel process network setup, scheduling and placement, communication handling, load balancing). Therefore the programmer may concentrate his efforts on the qualitative aspects of parallelism exploitation rather than on the cumbersome, error prone, implementation details mentioned above [14].

Currently available skeleton programming environments requires considerable programming activity in order to implement the skeleton based programming languages [3, 5]. Skeleton programs are compiled by generating the code of a network of cooperating processes out of the programmer skeleton code, and providing all the code necessary to place, schedule and run the processes on the processing elements of the target architecture. When compiling skeleton programs, the knowledge derived from the skeleton structure of the user program is exploited via heuristics and proper algorithms. Eventually, efficient code is obtained, exploiting the peculiar features of the target architecture at hand [14].

[1] This work has been partially funded by the Italian MURST *Mosaico* project.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1175–1184, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Capitalizing on the experience we gained in the design of such environments [2, 8], we designed a library (**SKElib**) that allows the programmer to declare skeletons out of plain C functions, to compose such skeletons and to command their evaluation by a simple C library call statement. The library allows the programmer to structure parallel computations whose pattern do not correspond to a skeleton by using standard Unix mechanisms. The library can be used on any cluster of workstations (COW) running Unix OS. This kind of architecture is commonly available and delivers very high performance at a price which is a fraction of the price of other, specialized, (massively) parallel architectures [4]. In this paper we first describe the choices taken in the library design (Sec. 2), then we discuss the implementation details of **SKElib** (Sec. 3) and eventually we present the performance results we achieved when running skeleton programs written with **SKElib** on a workstation cluster (Sec. 4).

2 Library Design

The skeleton set provided to the parallel application programmer by **SKElib** includes a small number of assessed skeletons [1, 16, 15]:

`farm` a task farm skeleton, exploiting parallelism in the computation of a set of independent tasks appearing onto the input stream.
`pipe` a skeleton exploiting the well known pipeline parallelism.
`map` a data parallel skeleton, exploiting simple data parallel computations and, particularly, computations where a result can be obtained combining a set of partial results. These results, in turn, are computed by applying a given function on all the data items obtained by partitioning an input data item into a set of independent data partitions.
`while` a skeleton modeling iterative computations.
`seq` a skeleton embedding sequential code in such a way that the code can be used as a parameter of other skeletons.

The skeletons we provided in **SKElib** can be nested, e.g. a `pipe` may have a `farm` stage that, in turn, computes tasks by exploiting pipeline parallelism. Such a skeleton set allows significant parallel applications to be developed and, in the meanwhile, is simple enough to be reasonably handled[2]. All these skeletons are provided to the programmer by including functions in the library that can be called to *declare* both sequential code to be used as a skeleton parameter (i.e. as a pipeline stage, via the `SKE_SEQ` function) and `pipe`, `farm`, `map` and `while` skeletons having other skeletons as parameters (`SKE_PIPE`, `SKE_FARM`, `SKE_MAP` and `SKE_WHILE` functions).

A different function is implemented in the library to *ask the evaluation* of a skeleton expression (`SKE_CALL`). This function takes parameters denoting where the input stream (the sequence of input data sets) and the output stream (the

[2] one of our main aims was to demonstrate the feasibility of the library approach to skeleton implementation

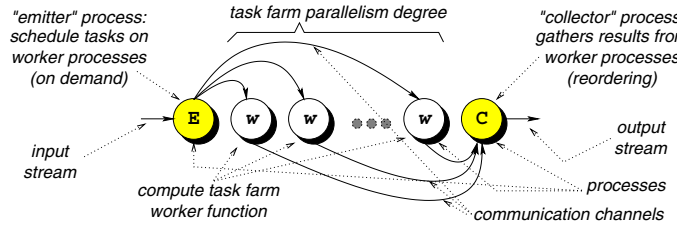


Fig. 1. Implementation template of the farm skeleton

sequence of the data results) have to be taken/placed. Such parameters, formally denoting the skeleton input and output streams, are plain Unix files. The programmer can ask the library to evaluate a skeleton expression taking input data from a file and delivering output data to another file. But he can also ask **SKElib** to evaluate the same expression by taking input data from the output of an `exec`ed Unix process whose output has been redirected to a named pipe, just by using the named pipe as the skeleton input stream. Further parameters must be supplied to `SKE_CALL` denoting the processing elements (PE) that can be used in the parallel skeleton computation.

We choose to implement skeletons by exploiting implementation template technology [2, 14]. Following this approach, each skeleton is implemented by setting up a network of communicating sequential processes arranged as described in a *template library*. This library holds an entry for each skeleton supported. The entry completely describes the structure of an efficient process network that can be used to implement the skeleton on the target architecture (i.e. the entry describes how many processes must be used, where the processes have to be placed, how they have to be scheduled, etc.). As an example, Figure 1 shows the process network relative to the implementation template of the `farm` skeleton used in **SKElib**. This template (as the other templates used in **SKElib**) does not maintain the input/output ordering of tasks. Results relative to input task i may appear on the output stream *after* the results relative to input task $i+k$ ($k > 0$). **SKElib** just preserves ordering between *application* input and output streams: results are delivered (using a proper buffering algorithm) on the output stream in the same order as the corresponding input data appeared onto the input stream.

SKElib uses plain TCP/IP Unix sockets to implement interprocess communications. We could have used any one of the communication libraries available in the Unix environment (e.g. MPI [13]), but the usage of such kind of libraries can prevent the user from using some of the typical Unix mechanisms and one of our goals was to provide a seamless skeleton framework.

The overall goal we achieved with **SKElib** has been to make available to the Unix programmer a new paradigm to structure common parallelism exploitation patterns while leaving the programmer the ability to "hand code" those parts of the parallel application that do not fit any of the skeletons in the library. It's worth pointing out that in classical skeleton frameworks the programmer is com-

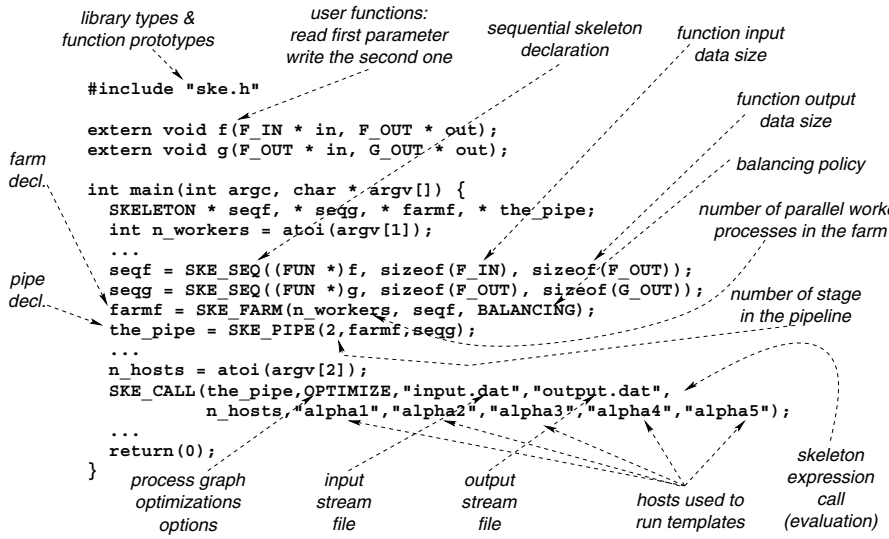


Fig. 2. Sketch of sample application code

pletely and consistently assisted in the implementation of parallel applications whose parallel structure fits some particular (nesting of) skeleton(s), but has no way to implement some parallel exploitation pattern which is not captured by the skeletons provided by using the other typical mechanisms supported by the hardware and software architecture at hand.

3 SKElib Implementation

Figure 2 sketches a simple parallel application written using **SKElib**. The application computes a stream of results out of a stream of input data. Each input data is used to compute a partial result via the function `f` and this partial result is used to compute the final result via the function `g`. This computation structure is naturally modeled by the `pipe` skeleton. By taking into account that the computation of function `f` is much more expensive than computation of function `g`, the programmer has inserted a `farm` in the first pipeline stage. Therefore, the program will read – in a cycle – an input data set from the input stream (file `input.dat`, actually), deliver it to a worker process of the `farm` that computes an intermediate result applying the `f` function, and eventually deliver such intermediate result to the process sequentially computing the `g` function.

The process leading to the execution of this skeleton program is outlined in Figure 3. First of all the program is compiled and linked to **SKElib** code. Then the user runs the program on a node of the target COW. When the control flow reaches the request for skeleton code evaluation (the `SKE_CALL` library call), the library code analyzes the skeleton structure declared by the programmer and

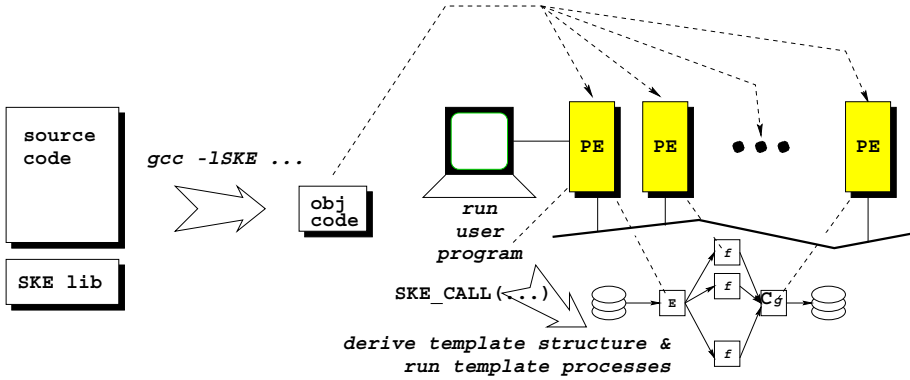


Fig. 3. Parallel application execution schema

"`execs`" on the processing elements of the COW all the processes necessary to implement the process network derived from the skeleton code.

Functions representing the sequential code executed by skeletons must be supplied as void C functions with two parameters: a pointer to the input data and a pointer to the output data. This restriction on the sequential code format has been introduced to avoid a number of unnecessary buffer copy operations in the library code, while requiring a moderate "extra" programming effort to the user.

The processes in the implementation templates of skeletons must be able to execute user supplied code and, in particular, the code embedded in the `seq` skeletons and all the related code. In order to allow the template code to execute such user code without charging the programmer with the task of following particular formats in the source code, we decided to exploit an SPMD execution model. **SKElib** wrappes the user `main`. Therefore, when the user program is run the library `main` function is executed in place of the user `main`. If the first command line argument is not the special (reserved) `SKEslave` string, the **SKElib** `main` just calls the user `main`. Otherwise **SKElib** takes the complete control of the execution and proper process templates are run instead of the user `main`, depending on the other command line parameters. Therefore, when the user runs the skeleton program supplying his own command line parameters, his `main` code is executed. When the user code reaches a `SKE_CALL` statement, the skeleton program is remotely run on the PEs in the COW. The command line parameters passed to the remote `exec` command are such that the **SKElib** `main` takes definitely the control and calls those parts of the **SKElib** code that implement the processes that must be run on the remote nodes to implement the skeleton templates. Which template code has to be executed on the nodes is derived by the `SKE_CALL` code by looking at the information gathered with skeleton declarations. This leads to an SPMD execution model of the skeleton program. The same program is eventually run on all the nodes of the target architecture. The behavior of the program on each node depends on the command

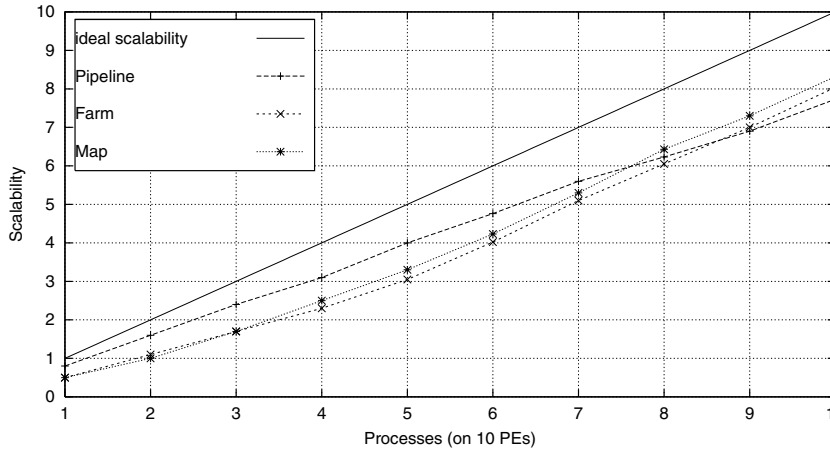


Fig. 4. Scalability results of the library

line parameters used to invoke the code. Parameters passed to the code executed on the remote nodes are both symbolic and pointer parameters. The symbolic parameters tell **SKElib** `main` which process templates are to be executed. The pointer parameters tell the library which user code must be eventually be called by the process templates. Pointer parameters (virtual addresses, actually) can be used because all the nodes eventually will run the same code – the one derived from the compilation of the user code linked with the library code.

This mechanism allows any user function to be called on the processing elements of the target architecture without requiring the user to supply the code embedded in the `seq` skeletons in a particular file/library. In the normal SPMD model, all the processes running on the different processing elements and executing the same code are started at the same time. In our case, one process is started by the user, whereas the other processes are started in the `SKE_CALL` code, via `rsh` calls. This approach has a drawback: all the user code is actually replied on all the processing elements participating in the computation. In case the user code is large, this may lead to virtual memory waste on the target architecture PEs.

The programmer may ask **SKElib** to optimize the object process graph by specifying a proper parameter in the `SKE_CALL` (in the example of Figure 2 an `OPTIMIZE` parameter is passed to this purpose). The optimizations currently performed by **SKElib** mainly concern process grouping on the same processing element. As an example, in a two stage pipeline with both stages exploiting farm parallelism, the collector process of the first `farm` and the emitter process of the second one are actually merged when the `OPTIMIZE` parameter is specified in the `SKE_CALL`. The resulting process network shows less communication overhead than the original one. We are considering further process graph optimizations. As an example, when the processing elements of the target architecture have to be multiprogrammed (due to the large number of processes derived from the user

code) communicating processes will be mapped to the same processing element, in such a way that a lower network overhead is paid.

Overall, **SKElib** has been implemented on a Linux COW using standard Unix mechanisms, namely `rsh` to run processes on remote processing elements and TCP/IP BSD sockets to perform inter-process communications. As a consequence, the library can be used on any workstation network supporting these mechanisms. The only thing to do in order to port the library across different COWs is to compile the library code. Due to the SPMD execution model adopted, however, all the processing elements in the COW must have the same processor (i.e. the same instruction set) as well as the same operating system. This implies that the library is not to guaranteed to work on a COW with processing elements running different (versions of the) operating system.

4 Experimental Results

We run some experiments to test **SKElib** performance. The experiments concerned both the evaluation of the absolute performance achieved by **SKElib** and a comparison with the performance values achieved by using Anacleto, the P3L compiler developed at our Department [8]. P3L is the skeleton based parallel programming language developed in Pisa since 1992 [2]. Anacleto compiles P3L programs generating C + MPI [13] code that can be run on a variety of parallel machines and workstation clusters.

All the experiments have been performed using Backus as the target architecture. Backus is a Beowulf class workstation cluster with eleven PC based workstations. Each PC sports a 233Mhz Pentium II with 128Mbyte of main memory, 2Gbyte of user disk space and a 100Mbit Fast Ethernet network interface card. The PCs run Linux (kernel 2.0.35) and are interconnected by a 3Com SuperstackII 100Mbit Ethernet switch. All the PCs are "dedicated" in that no other computations where performed on the machines during the experiments but our skeleton processes and the usual system processes.

When executing skeleton code on the COW, we achieved effective speedups with respect to the execution of the corresponding sequential code. Concerning scalability, the typical results we achieved are summarized in Figure 4. The Figure plots the scalability[3] s achieved in the execution of medium to coarse grain skeleton programs written using our library and exploiting parallelism by using a single skeleton. The x-axis is relative to the number of processes used to implement the program. The pipeline program used a 10 stage `pipe`. The `farm` and `map` have been compiled to use a number of processes ranging from 1 to 20. **SKElib** is able to schedule more than a single template process on a single processing element (currently the scheduling is performed with a round-robin algorithm) in such a way that communication and computation times relative to different processes can be overlapped and therefore a certain degree of communication

[3] $s = \frac{T(1)}{T(n)}$, where $T(i)$ is the time spent in computing the parallel program using i PEs

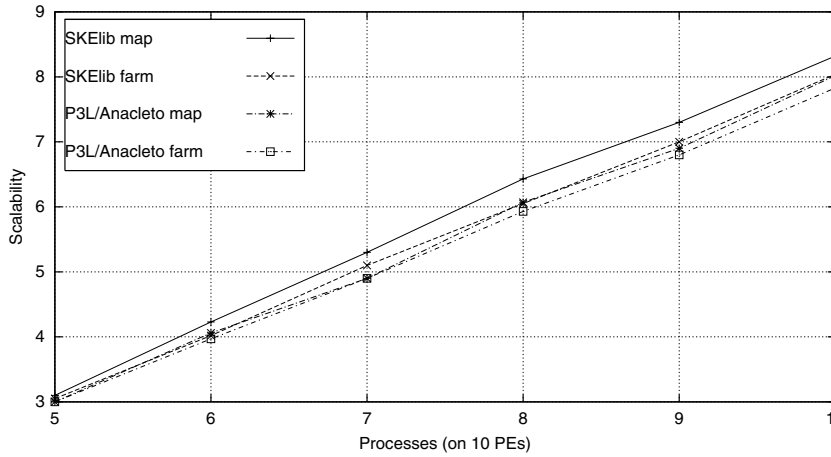


Fig. 5. Performance: **SKElib** vs. Anacleto

latency hiding is achieved. As a consequence, the runs requiring more than 10 processes perform node multiprogramming on the COW PEs. From Fig. 4 it is clear that the skeletons implemented in the library achieve quite a good efficiency. Using a number of processes that do not exceed the number of processing elements available, we achieve an efficiency which is more that 80%. However, as the library has been designed to take advantage of the node multiprogramming facilities offered by Unix, we also run skeleton programs implemented with a higher number of processes than the actual number of processing elements in the machine. In this case we achieved scalability values larger that 9.8 on 10 PEs. Such results where achieved running `map` and `farm` skeletons using up to 20 processes. It's worth pointing out that in order to increase the parallelism degree of a skeleton – and, as a consequence, the number of processes used to implement the skeleton – the only thing the programmer must do is to specify the proper parameter value in the declaration of the skeleton. In the code of Figure 2, the programmer specifies that the `farm` must have a parallelism degree of `n_workers` just passing this number as the first parameter of the `SKE_CALL` function call.

Figure 5 shows the results achieved by our library and the results achieved running similar skeleton programs written in P3L and compiled using Anacleto. The performance Figures are similar, but our library sports slightly better results. This is in part due to the fact that Anacleto generates C + MPI code and therefore communications are performed via MPI primitives that, in turn, are implemented on top of the same sockets we directly used to implement **SKElib** communications. However the result is quite satisfactory. **SKElib** presents some advantages with respect to the use of P3L/Anacleto, in particular concerning the possibility to use mixed skeleton/standard parallelism exploitation mechanisms, and still delivers a performance which is slightly better than the one delivered by the Anacleto runs.

All the results discussed above have been achieved by using either simple programs using a single parallelism exploitation pattern/skeleton (just to test the different skeleton implementation features) or simple applications templates (image processing, simple numerical algorithms such as matrix multiplication or mandelbrot set computation). In this way we have been able to validate the **SKElib** design and to draw complete performance figures. Currently we are developing full applications in order to better evaluate both the expressive power provided to the programmer by **SKElib** and the performance achieved when exploiting complex parallelism exploitation patters using significant skeleton compositions.

5 Related Work & Conclusions

Many efforts are being made to implement skeleton based parallel programming environments. Our research group in Pisa is active in the development of SkIE [16, 3]. Serot is currently developing a skeleton framework for image processing [15]. Darlington's group at the Imperial College in London also gave substantial contributes to skeleton research [11] as well as the group of Burkhart [7]. [5] discusses a skeleton framework mainly focused on data parallelism exploitation. [6] studied skeletons in a functional programming framework and we also provided a "skeleton embedding" within Ocaml, which is the ML dialect of INRIA [10]. All these projects (but the one described in [10], that actually supports skeletons in Ocaml via a library) either provide skeletons as new programming languages and/or language extensions or they rely on a heavy compile process in order to compile skeleton programs to parallel object code. There is no previous attempt to provide a skeleton programming environment via a plain C library, to our knowledge. Anyway, we exploited experiences performed in all these projects in the design of the template system used to implement the skeletons in **SKElib** (in particular the approach used to implement skeletons in **SKElib** is almost completely derived from the projects discussed in [2, 11]).

Different project exists, not directly involved with the skeleton research track, aimed at making available libraries providing the programmer with suitable tools to structure parallelism exploitation within applications. As an example, [12] discusses a library allowing data parallel computations to be expressed using the classes provided within a C++ library. We owe these projects, as well as the project summarized in [10], for some of the ideas used to implement data parallel skeleton templates and the general library structure.

In this work, we discussed the design and implementation of **SKElib**, a library providing the C Unix programmer with a simple way to implement parallel applications using skeletons. **SKElib** has been developed on a Linux COW. Using **SKElib**, parallel applications can be written that exploit parallelism both by using skeletons and by using standard Unix concurrency mechanisms. Skeletons can be used to program those application parts that match standard parallelism exploitation patterns. Standard Unix mechanisms can be used to program those application parts that do not match the skeleton provided. We showed exper-

imental results demonstrating that the approach is feasible in terms of performance. We also showed that the performance results achieved by **SKElib** are slightly better than the results obtained by using existing skeleton compilers.

References

[1] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[3] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[4] M. Baker and R. Buyya. Cluster Computing at a Glance. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, pages 3–47. Prentice Hall, 1999.

[5] George Horatiu Botorog and Herbert Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

[6] T. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, 1994.

[7] H. Burkhart and S. Gutzwiller. Steps Towards Reusability and Portability in Parallel Programming. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 147–157. Birkhauser, April 1994.

[8] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proceedings of the PCW'97*, 1997. Camberra, Australia.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[10] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel Functional Programming with Skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.

[11] Darlington, J. Guo, Y.K, H.W. To, and Y. Jing. Functional skeletons for parallel coordination. In *Proceedings of Europar 95*, 1995.

[12] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the Parallel Standard Template Library. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 7–11, July 1997.

[13] M.P.I.Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[14] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[15] J. Serot, D. Ginhac, and J.P. Derutin. SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In *Proceedings of the 5th International Parallel Computing Technologies Conference (PaCT-99)*, September 1999.

[16] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency*, 6(4):68–73, 1998.

Token-Based Read/Write-Locks for Distributed Mutual Exclusion

Claus Wagner[1] and Frank Mueller[2]

[1] Computer Science Department, TECHNION, Haifa, Israel 32000
[2] Humboldt University Berlin, Institut f. Informatik, 10099 Berlin, Germany
phone: (+49) (30) 2093-3011, fax: -3010
mueller@informatik.hu-berlin.de

Abstract. The contributions of this paper are twofold. First, a protocol for distributed mutual exclusion is introduced using a token-based decentralized approach, which allows either multiple concurrent readers or a single writer to enter their critical sections. This protocol utilizes a dynamic structure incorporating path compression to keep the messages overhead low resulting in an average complexity of O(log n) messages per request. Second, this protocol is evaluated in comparison with another protocol that uses a static structure instead of dynamic path compression. The measurements show that although concurrent readers may require at most one additional message per entry, the concurrent execution of critical sections results in faster responses of up to 30% for short critical sections. For longer critical sections, savings in the overall execution time increase with the fraction of readers to up to 50%. In particular applications with large fractions of readers, *e.g.*, database queries, may exploit these benefits. The results further indicate that problems with fine-grained parallelism are more suitable for the dynamic protocol proposed here while the static protocol used for comparison performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

1 Introduction

Common resources in a distributed environment may require that they are used in mutual exclusion. This problem is similar to mutual exclusion in a shared-memory environment. However, while the shared memory architectures generally provide atomic instructions (*e.g.*, test-and-set) to implement mutual exclusion, such provisions do not exist in a distributed environment. Furthermore, commonly known mutual exclusion protocols for shared-memory environments that do not rely on hardware support still require access to shared variables.

In distributed environments, mutual exclusion is provided via a series of messages passed between nodes that are interested in a certain resource. Several protocols to solve mutual exclusion for distributed systems have been developed [1]. They can be distinguished by their approaches as token-based and non-token-based. The former ones may be based on broadcast protocols or they may use logical structures with point-to-point communication.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1185–1195, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In this work, we assume a fully connected network (complete graph). Network topologies that are not fully connected can still use our protocol but will have to pay additional overhead when messages from A to B have to be relayed via intermediate nodes. Second, we assume reliable message passing (no loss, duplications, or modifications of messages) but we do allow out-of-order message delivery with respect to a pair of nodes, *i.e.*, if two messages are sent from node A to node B, then they may arrive at B in a different order than they were sent. Our assumption is that local operations are several orders of a magnitude faster than message delivery since this is the case in today's networks and the gap between processor speed and network speed still seems to widen. Thus, our main concern is to reduce the amount of messages at the expense of local data structures and local operations.

We introduce a new protocol to provide mutual exclusion in a distributed environment that distinguishes read and write requests. The protocol is based on some ideas by Naimi *et al.* [10], *i.e.*, a decentralized protocol that did not distinguish readers and writers. We explain the design of our approach, illustrate it by examples and provide the pseudocode for it. Measurements in a simulation testbed are obtained to evaluate the performance of our protocol.

2 Related Work

A number of protocols exist to solve the problem of mutual exclusion in a distributed environment. Chang [1] and Johnson [4] give a more detailed overview and compare the performance of such protocols. Non-token based protocols establish consensus between nodes before entry to a critical section is granted and often employ global logical clocks and timestamps [6]. Token-based protocols link permission of entry to token ownership. The broadcast-based subclass of protocols does not require specific communication paths since all nodes always receive a message. Token-based protocols with logical structures constrain messages to certain paths, which may be static or they may change dynamically. Our work focuses on this last class of protocols since their message complexity of O(log n) is lower than that of most of the above approaches, which require at least O(n) messages on the average for a request in a network of n nodes. Exceptions are Maekawa with $O(\sqrt{n})$ realized by an evenly distributed set of nodes over their quorum (subset of nodes that have to grant access to a requester) [8] and Kumar with $O(n^{0.63})$ building on the former approach [5]. Raymond proposed an O(n) quorum protocol [12], which was improved by Srimani and Reddy [14]. However, quorum protocols with k concurrent entries to the critical section do not provide the distinction between an arbitrary number of readers and writers of our work.

Another protocol by Raymond [13] utilizes a static tree for request propagation and utilizes local queues for requests. This work was extended by Neilsen and Mizuno [11] to replace these local queues by a distributed queue. In contrast, Chang, Singhal and Liu [2] use a dynamic tree similar to Naimi *et al.* [10]. The protocol makes use of path compression similar to independent work on memory consistency protocols [7]. Our work was influenced by the approaches

of Raymond and Naimi *et al.*. Our protocols are based on the same assumptions and use similar data structures for write requests, which also ensures that the complexity of the original protocols is preserved in our solution in the absence of read requests. The additional support of concurrent readers in our protocols then only adds a constant overhead of one message for a read chain, if this chains build up. Hence, the order of complexity of our protocols in terms of the number of messages remains the same as within the original protocols.

3 Dynamic Reader/Writer Protocol for Mutual Exclusion

The dynamic protocol utilizes a directed tree-like structure. The edges form a chain leading new requests from node R to the last requester L (or the token holder if no requests were pending). While a request is in transit, intermediate nodes set their edges to point to the new requester R, thereby providing path compression, *i.e.*, future requests will propagate directly to R from any of the intermediate nodes similar to Naimi *et al.* [10].

An example is depicted in Fig. 1 where a (read) request from A is sent via B to T. The intermediate nodes B and T have edges directed at A afterwards. If T is still engaged in the critical section with a write lock, the read request cannot be served yet. Instead, T creates a next pointer (dashed edge) to A indicating the next recipient of the token. Once T exits its critical section, it sends the token to A and removes the next edge. A proceeds with its critical section under read protection. At this time, C issues a read request that is sent to A via B resulting in new edges from A and B to C. A responds by sending the token to C even before exiting its critical section because both A and C may execute concurrently in their critical sections under read protection. In addition, A registers C as the next reader (dotted edge) and C notes the fact that it is the last reader in a chain of readers. Node C will hold on to the token until (1) a request from another reader arrives next or (2) C and A exit their critical sections. The former case allows more concurrent readers to be served if they arrive ahead of writers. The latter case ensures that a writer will only be served after all reads have completed. Once A exits its critical section, it sends an acknowledgement to C, which must be received by C before the token may be forwarded to a writer.

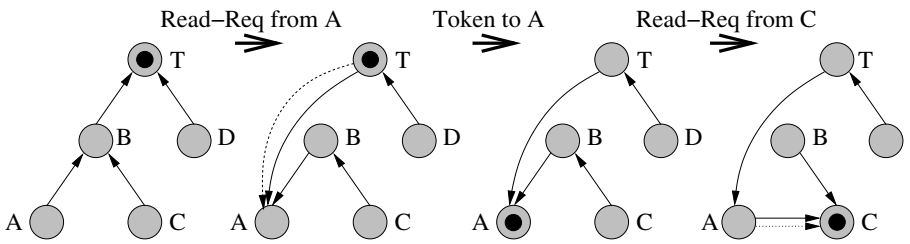


Fig. 1. Simple Example

The example illustrates that the protocol requires directed edges that requests travel along. It includes a distributed next queue of pending requests whose source is the token holder while the sink is the last requester. A distributed read queue links concurrent readers starting at the earliest requester still engaged in a critical section via consecutive requesters to the sink of the reader chain. Consecutive members of the read chain are either in the critical section or have not received an acknowledgement from their predecessor in the read chain yet. When a node issues multiple read requests in a row, the read chain may be circular imposing the necessity to log pending recipients of acknowledgements in a local FIFO queue. Requests are always handled in the order that they arrive at the tail of the request queue, *i.e.*, a write request always terminates a chain of readers and subsequent read requests are served after the write. This ensures fairness and avoids starvation. The following notation is used in order to reflect these requirements when describing the protocol:

Token is true if nobody is engaged within a critical section.

Dir represents an edge that points towards the last requester or, in the absence of requests, to the token holder, creating a distributed tree (solid edges).

Next points to the next requester of the token. The next chain represents a distributed queue of pending requests (dashed edges).

Next_readers represents a FIFO list of readers with outstanding acknowledgements. The next readers over all nodes form a chain of concurrent readers that will be reduced successively by acknowledgements (dotted edges).

Pending_Acks counts the number of acknowledgements a node still expects. When positive, a node may only forward the token, if it possesses it, to a reader since other readers are still active. Only when the value because zero, may the token be forwarded to a writer.

Token_mode is write if a writer is engaged in a critical section. The value is read if the last reader (of the reader chain) is in a critical section or multiread if the token has already been sent to a concurrent reader node. It is undef in any other case.

Next_mode is read/write if next points to the next R/W requester, respectively.

The pseudocode of the protocol is depicted in Fig. 2. Write requests are handled similar to the protocol by Naimi *et al.* [10]. Next, the relation between the abstract operational model and the actual implementation shall be discussed.

At initialization, all nodes point via dir to the start node, which is the token holder. Notice that the edge of the token holder to himself is omitted in the example since it represents a special case beyond the tree structure. A lock request for a locally unused token can be served right away. All other requests propagate along the dir edges while the requester clears his dir pointer since he is the last requester waiting for the token to arrive. Once the token arrives, the flag to indicate if the node should "expect an acknowledgement" is set according to the value piggybacked with the token message and the number of pending acknowledgements may be incremented accordingly. In case of concurrent reads, the token is forwarded to the next reader and the mode is set to multiread.

```
token = (self == start); // TRUE if token locally available
dir = start;             // edge with changing destination for requests propagation
next = NULL;             // next receiver of token (⇒ dist. Q of unserved requesters)
next_readers = φ;        // FIFO list of next readers (⇒ dist. Q of conc. readers)
pending_acks = 0;        // number of outstanding acknowledgements
token_mode = undef;      // protection of critical section (undef/read/multiread/write)
next_mode = undef;       // protection for next receiver (undef/read/write)
PROC lock(mode) IS
    IF ¬ token THEN
        SEND request(self, mode) to dir;
        dir = self;
        AWAIT(token(&expect_ack));
        IF expect_ack THEN
            pending_acks += 1;
        ENDIF;
        concurrent_read = (mode == read
            AND next_mode == read);
        IF next ≠ NULL AND
            concurrent_read THEN
            SEND token(TRUE) to next;
            append(next_readers, next);
            next = NULL;
            token_mode = multiread;
        ELSE
            token_mode = mode;
        ENDIF;
    ELSE
        token_mode = mode;
    ENDIF;
    token = FALSE;
END lock;

PROC unlock IS
    IF token_mode == multiread AND
        pending_acks == 0 THEN
        SEND ack to head(next_readers);
        delete_head(next_readers);
    ENDIF;
    IF next ≠ NULL AND
        pending_acks == 0 THEN
        SEND token(FALSE) to next;
        next = NULL;
    ELSE IF token_mode ≠ multiread THEN
        token = TRUE;
    ENDIF;
    token_mode = undef;
END unlock;

PROC receive_request(sender, mode) IS
    IF dir ≠ self THEN
        SEND request(sender, mode) to dir;
    ELSE
        concurrent_read = (token_mode ==
            read AND mode == read);
        IF token AND pending_acks == 0
            OR concurrent_read THEN
            SEND token(¬ token) to sender;
            token = FALSE;
            IF concurrent_read THEN
                token_mode = multiread;
                append(next_readers, sender);
            ENDIF;
        ELSE
            next = sender;
            token = FALSE;
            next_mode = mode;
        ENDIF;
    ENDIF;
    dir = sender;
END receive_request;

PROC receive_ack IS
    pending_acks -= 1;
    IF pending_acks > 0 OR
        ¬ empty(next_readers) AND
        token_mode == undef THEN
        send ack to head(next_readers);
        delete_head(next_readers);
    ELSE IF token AND pending_acks == 0
        AND next ≠ NULL THEN
        send token(FALSE) to next;
        next = NULL;
        token = FALSE;
    ENDIF;
END receive_ack;
```

Fig. 2. Pseudocode of Read/Write-Lock Protocol

Otherwise, the request mode is stored before entering the critical section. The token flag is also cleared during the critical section indicating that it is in use.

Upon the end of the critical section (unlock), several cases are distinguished. Readers in the chain (except for the tail of the chain) send an acknowledgement to the next member of the chain, thereby reducing the chain if they have already received an acknowledgement from their predecessor. If a next requester exists and all read requests have completed (no pending acknowledgements), the token is sent to the next requester. Otherwise, the token is marked as locally available unless it was already sent to a concurrent reader at an earlier time.

A receiver of a request from some sender also has to distinguish certain cases before changing its dir edge to the sender. If the receiver is not the last requester, then he simply forwards the request with the sender's id along the directed edges. Otherwise, one of two cases may apply. If the token is available and all readers have completed (no acks pending) or if the request is for a concurrent read, the receiver sends the token to the sender, clears the token status and records the next concurrent reader if necessary. The piggybacked flag is true if the new request represent a concurrent read. If neither the token was available nor was the request for a concurrent read, the sender is logged as the next requester.

Upon reception of an acknowledgement, the number of pending acknowledgments is decremented and an acknowledgement is sent to the next reader if this had not already been done in the unlock operation. Otherwise, the token is forwarded to the next requester if all acknowledgements have been sent and the read chain has collapsed. Notice that the next requester was either a writer or a reader at the head of another read chain so that the piggybacked value FALSE requires no checks for acknowledgments by this requester (similar as in unlock).

The protocol ensures mutual exclusion since either (a) a single writer may own the token or (b) the tail of the read chain owns the token (and predecessors in the read chain may execute their critical read section concurrently). Deadlocks are avoided since requests are forwarded via a tree-like structure to the tail of the requester queue. These requests are then served in the order that they were queued according to the mutual exclusion property, *i.e.*, circular dependencies cannot occur. Starvation is avoided by the FIFO policy of queuing requests, *i.e.*, a new request will be served exactly after all its predecessors in the FIFO queue have been served (again subject to mutual exclusion properties for readers/writers). The protocol has a complexity of O(log n) messages since it uses the same tree structure and messages for requests as Naimi *et al.* [10] although a constant of one message per reader in a chain is added.

4 Experimental Platform

The experiments were conducted in a simulation environment where distributed nodes were mapped onto processes and communication was performed via asynchronous TCP operations on a uniprocessor. Each process is itself multi-threaded to allow asynchronous communication. Two components implemented as threads require further explanation: A user thread may execute application code includ-

ing lock and unlock operations. The communication server receives requests, tokens and acknowledgements. Requests are serviced right away while the other messages are registered. If a user thread awaits such a message, he is activated. Otherwise, the message is stored and when the user thread performs the next await operation, this operation will not block. These operations are implemented using a POSIX mutex and condition variables. In addition, each operation of the protocol depicted in Fig. 2 is executed within mutual exclusion, *i.e.*, the protocol ensures a monitor semantics to keep its internal state consistent, which arbitrates the threads on a node. The monitor is realized by the POSIX mutex also used in conjunction with the condition variables. In prior experiments, the testbed has proved its adequacy to evaluate different mutual exclusion protocols yielding results comparable to prior work [3].

The main experiment consists of a synthetic benchmark as a user thread along the lines of prior work [3, 9]. A loop of i iterations consists of a critical section followed by non-critical code. Durations of the critical section ξ and of non-critical code Γ can be varied to simulate different request rates. Requests are randomized by $\pm 10\%$ around ξ and Γ providing a framework of contention where requests are clustered around a certain point in time but arrive in random orders, as previously shown in [15]. The number of distributed nodes N can be varied and the fraction of readers may be specified to evaluate the protocol.

This setup allows the measurement of various aspects. First, the number of messages exchanged per entry (NME) of a critical section can be determined. This allows a comparison between different methods and parameters to relate timing results to message overhead. Second, the average response time for a series of requests is calculated. This provides the means to determine savings per critical section entry. Third, the overall execution time of a test run may be measured. This indicates the overall savings that may be obtained in an application. Varying the proportion of readers and writers gives an insight on how a protocol performs for a varying number of read requests. But it also provides a comparison with the base protocol without distinction between readers and writers, namely when the fraction of read locks is set to 0.

The testbed includes the implementation of the dynamic tree-structured protocol for token-based mutual exclusion in a distributed environment, called DVARW in the following. In addition, a token protocol based on a static tree structure has also been implemented. The base protocol by Neilsen and Mizuno [11] has been extended to distinguish read and write locks [15] and is referred to as SVARW in the following. Neilsen and Mizuno allow requests only to travel along the edges of the static tree. An edge may reverse its direction. Thereby, edges form a chain pointing to the last requester. In contrast, Naimi's approach adopted in this work modifies the sink of edges, not just their direction, as requests propagate along paths. This results in path compression intended to reduce the number of hops for future requests. The modifications to Neilsen and Mizuno's algorithm to distinguish readers and writers are similar to the protocol described in this paper including token forwarding for concurrent reads before completing a critical section and sending acknowledgements along the chain of

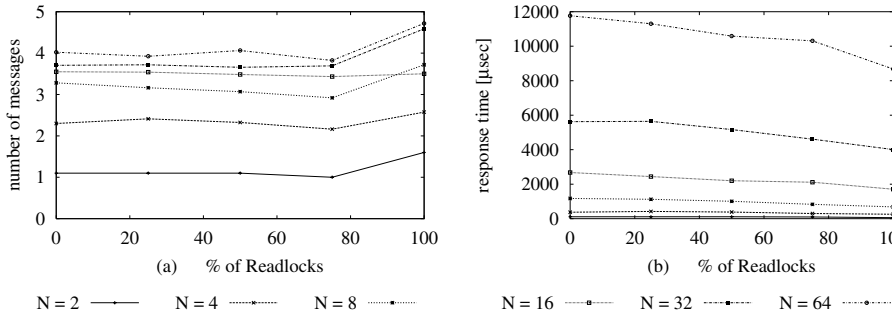


Fig. 3. Messages and Response Time of the Dynamic Tree Protocol (DVARW)

concurrent readers upon completion of the critical section. The details of the static reader/writer protocol are beyond the scope of this paper (see [15]).

5 Measurements

The measurements were obtained on a 133MHz Pentium under Linux 2.1 executing the benchmark described in the aforementioned testbed. The number of iterations was $i = 30$. The duration of a critical section was chosen as $\xi = 200\mu sec$ while the non-critical part was $\Gamma = 2000\mu sec$. This choice reflects that critical sections tend to be short while non-critical code takes up more of the overall execution time. The choice of values was also influenced by prior simulation studies on the performance of distributed mutual exclusion protocols [3, 9].

Fig. 3 depicts the results for the dynamic tree protocol under varying number of nodes ($N = 2..64$) and varying fractions of read requests. Fig. 3(a) shows that the number of messages depends on the total number of nodes in the system, which is caused by an increasing path length to propagate requests. For a given number of nodes, the number of messages remains constant up to a certain point. This indicates that the dynamic protocol with read/write distinction performs just as well as the base protocol without this distinction, which underlines the qualities of our approach. At some point, read requests dominate (>75%) and the number of messages increases by almost 1 at 100%. This effect is caused by acknowledgement messages that are only sent when read chains start to build up at high reader frequencies. Fig. 3(b) indicates that the response times increases linearly with the number of nodes. Furthermore, with an increasing portion of read requests, the response time is reduced up to 30% for $N = 64$. The actual reduction (the gradient of each curve) increases with the number of nodes. This demonstrates that although concurrent readers may require an additional message, the concurrent execution of critical sections results in faster responses. In other words, applications can benefit from read concurrency with this protocol (quantified more precisely in Fig. 5).

In contrast, Fig. 4 depicts the results for the static tree protocol. The number of messages remains unchanged or increases slightly, depending on the number of nodes, since messages have to propagate along longer paths within the static tree. The response time also increases linearly for the number of nodes while it decreases with higher reader frequencies. While the response time of the static variant is higher than in the dynamic counterpart in the absence of readers, the results are about the same in the absence of writers. In the latter case, the number of messages is also about the same, which explains this behavior.

Fig. 5 summarizes the impact of a read/write distinction for an entire execution of the benchmark. Here, the number of iterations was $i = 20$, there were $N = 16$ nodes, the critical section was $\xi = 5000\mu sec$ and $\xi = 20.000\mu sec$ while non-critical code took $\Gamma = 2000\mu sec$. This simulates a case where complex operations in critical sections dominate non-critical code. Both the dynamic protocol in Fig. 5(a) and the static protocol in Fig. 5(b) show execution time savings of around 50% when all requests are for read access compared to only write requests. The savings are slightly higher for longer critical sections than for shorter ones. While some savings already materialize around 50% readers, the savings increase even more rapidly for a higher proportions of readers. Typically, applications experience far more read than write requests, *e.g.*, database queries. Hence, these savings have an impact on the overall performance.

The last experiment was conducted to determine the impact of different request frequencies on the protocol. For this purpose, Γ was varied for $N = 32$ nodes, $i = 20$ iterations and $\xi = 200\mu sec$ per critical section. Figure 6 shows that for high request frequencies (small Γ) the dynamic protocol results in fewer messages (about one messaged saved) and shorter response times (by about 10%) compared to the static case. For low request frequencies, the differences between the protocols diminishes. This is caused by the properties of path compression in the dynamic protocol where the structure of the dynamic tree degenerates into a list under these circumstances. This shows that the dynamic protocol exhibits its full advantage mostly for applications which use synchronization heavily throughout its execution. In short, problems with fine-grained

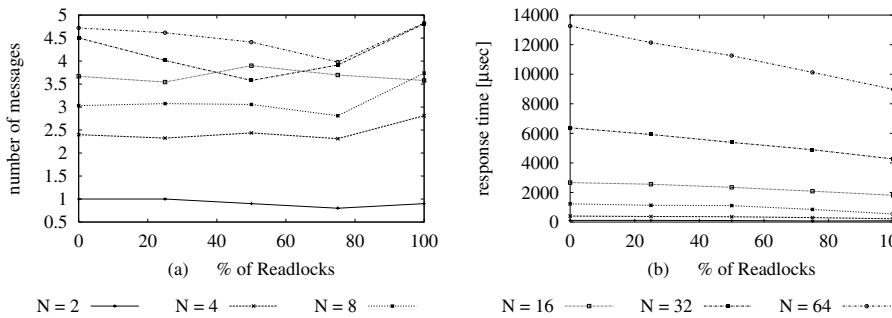


Fig. 4. Messages and Response Time of the Static Tree Protocol (SVARW)

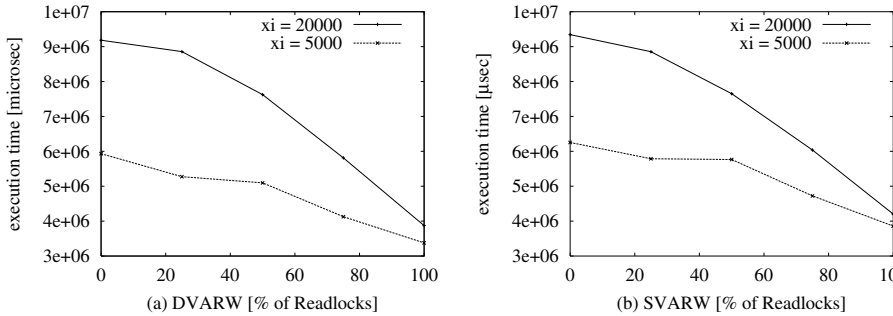


Fig. 5. Overall Execution Time (Dynamic vs. Static Tree)

parallelism are more suitable for the dynamic protocol while the static protocol performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

6 Conclusions

We developed a protocol for distributed mutual exclusion that distinguishes reader and writer requests utilizing a dynamic tree structure to reduce message overhead resulting in an average complexity of O(log n) messages per request. This protocol was evaluated in comparison with a protocol using a static structure. The measurements show that although concurrent readers may require at most one additional message per entry, the concurrent execution of critical sections results in faster responses of up to 30% for 64 nodes when critical sections are short. For longer critical sections, savings in the overall execution time increase with the fraction of readers to up to 50%. In particular applications with large fractions of readers, *e.g.*, database queries, may exploit these benefits. The results further indicate that problems with fine-grained parallelism

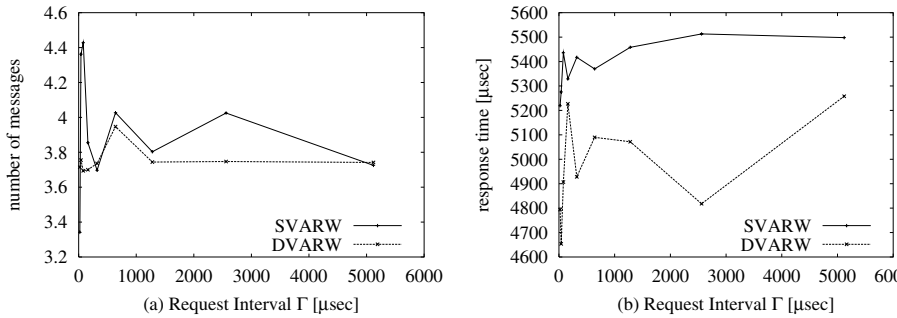


Fig. 6. Messages and Response Time for Varying Request Frequencies

are more suitable for the dynamic protocol proposed here while the static protocol used for comparison performs equally well for coarse-grained parallelism. Overall, reader/writer distinction provides promising benefits in both cases.

References

[1] Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, March 1996.

[2] Y. Chang, M. Singhal, and M. Liu. An improved O(log(n)) mutual exclusion algorithm for distributed processing. In *Int. Conference on Parallel Processing*, volume 3, pages 295–302, 1990.

[3] S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. In *International Parallel Processing Symposium*, pages 255–259, 1997.

[4] T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 258–264, 1995.

[5] A. Kumar. Hierachical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. On Computers*, 40(9):994–1004, 1991.

[6] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[7] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.

[8] M. Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[9] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[10] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.

[11] M. L. Neilsen and M. Mizuno. A dag-based algorithm for mutual exclusion. In *Distributed Computer Systems*, pages 354–360, 1991.

[12] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(4):189–193, February 1989.

[13] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Systems*, 7(1):61–77, February 1989.

[14] P. Srimani and R. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, January 1992.

[15] C. Wagner. Algorithmen zum gegenseiten Ausschluß in verteilten Systemen. Master's thesis, Humboldt University Berlin, Germany, December 1999.

On Solving a Problem in Algebraic Geometry by Cluster Computing*

Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
FirstName.LastName@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

Abstract. We describe a parallel solution to the problem of reliably plotting a plane algebraic curve. The sequential program is implemented in the software library CASA on top of the computer algebra system Maple. The parallel version is based on Distributed Maple, a parallel programming extension written in Java. We evaluate its performance on a cluster of workstations and PCs, on a massively parallel multiprocessor, and on a cluster that couples workstations and multiprocessor.

1 Introduction

We describe a parallel solution to the problem of reliably plotting a plane algebraic curve. Our starting point is the software library CASA (computer algebra software for constructive algebraic geometry) which has been developed on top of the computer algebra system Maple under the direction of the third author [2].

The basic objects of CASA are algebraic sets represented e.g. as systems of polynomial equations. Algebraic sets represented by bivariate polynomials model plane curves; an important problem is the *reliable* plotting of such curves. Numerical methods often yield qualitatively wrong solutions, i.e., plots where some "critical points" (e.g. singularities) are missing. For instance, the left diagram in Figure 1 shows a plot generated by the Maple function `implicitplot`. The numerical approximation fails to capture two singularities; even if we refine the resolution of the underlying grid, only one of the missing singularities emerges.

On the contrary, the CASA function `pacPlot` produces the diagram shown to the right. This plot of an algebraic curve a is generated by a hybrid combination of exact symbolic algorithms for the computation of all critical points and of fast numerical methods for the interpolation between them [3] (see Figure 1):

1. Compute the critical points of a in the y-direction and sort them according to their y-coordinates.
2. Intersect a with the horizontal lines that lie in the middle of the stripes determined by the y-coordinates of the critical points.
3. Trace a from each intersection point in both directions towards the border points of the stripe.

* Supported by the FWF projects P11160-TEC (HySaX) und SFB F013/F1304.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1196–1200, 2000.
© Springer-Verlag Berlin Heidelberg 2000

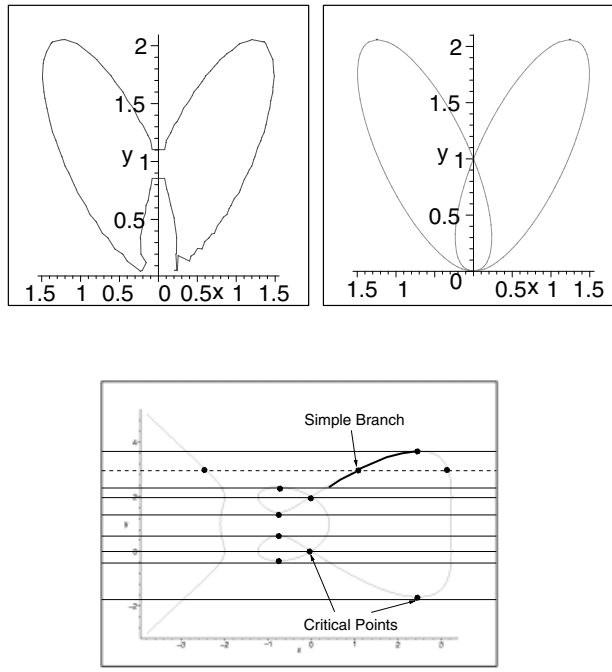


Fig. 1. Plotting an Algebraic Curve

`pacPlot` spends virtually all computation time in Step 1, a symbolic preprocessing phase that applies exact arithmetic to compute those points that determine the topological structure of a, essentially the singularities and the extrema in one coordinate direction. Figure 2 sketches the structure of the algorithm that computes a set of intervals that isolate these critical points.

The computation time of the critical point computation dominates the total plotting time and even on fast workstations only curves with total degree up to 8 or so can be plotted in a time that is acceptable for interactive usage. In the diploma thesis of the second author [1], this function has been parallelized.

2 The Parallelization Approaches

Investigating the dynamical behavior, it turns out that it does not pay off to parallelize the outermost loop which iterates over all systems in S: typically there are not more than two systems and only one system is not immediately recognized as trivial. Therefore we apply parallelism on several other levels:

1. parallel resultant computation,
2. parallel real root isolation,
3. parallel solution test,
4. parallel interval refinement.

```
P := criticalPoints(a(x, y)):
  P := ∅
  S := {(p(y), q(x, y)) |
    ∃p'(y) : (p'(y), q(x, y)) ∈ triangulize(a(x, y), ∂a(x,y)/∂x), p(y) ∈ factor(p'(y))}
  for (p(y), q(x, y)) ∈ S do
    r(x) := resultant_x(p(y), q(x, y))
    X := realroot(r(x))
    Y := realroot(p(y))
    for x ∈ X do
      q'(y) := squarefree(q(x, y), x.0, p(y))
      q''(y) := squarefree(q(x, y), x.1, p(y))
      for y ∈ Y do
        if test(q'(y), q''(y), y, p(y)) then P := P ∪ {(x, y)}
      end
    end
  end
end
```

Fig. 2. Computation of Critical Points

Parallel Resultant Computation A single resultant computation may take a good deal of the computation time of the whole algorithm. Our parallelization approach applies a modular method to compute the resultant in multiple homomorphic images of the domain and to combine the homomorphic results to the result in the original domain. We thus get a divide and conquer structure where both the divide phase (the modular resultant computation) and the conquer phase (the "Chinese Remaindering" computation) can be parallelized [4].

Parallel Real Root Isolation The isolation of the real roots of the resultant by Uspensky's method, a recursive divide and conquer search algorithm, may also take a significant computation time. A naive parallelization of this algorithm typically yields poor speedups due to the narrowness of the highly unbalanced search tree. A parallelization of the arithmetic in each recursive invocation step is only feasible for tightly coupled multiprocessors. Therefore our approach is to broaden and to flatten the search tree (to a certain extent) by applying speculative parallelism [1].

Parallel Solution Test and Interval Refinement The tests which of the candidates (x, y) are indeed solutions of the given system can be performed in parallel in a straight-forward fashion. Likewise, we can apparently refine all isolating intervals in parallel to the desired accuracy [1].

We have implemented all four ideas on the basis of Distributed Maple.

3 Distributed Maple

Distributed Maple is an environment for writing parallel programs on the basis of the computer algebra system Maple developed by the first author [4]. It allows to create tasks and to execute them by Maple kernels running on various machines of a network. Each node of a session comprises two components (see Figure 3):

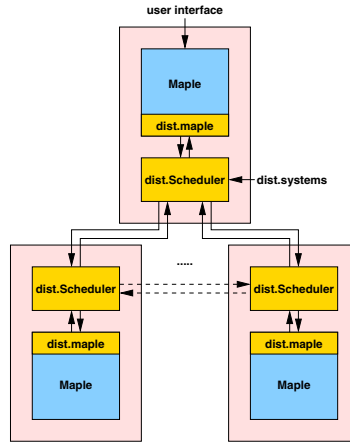


Fig. 3. Distributed Maple Architecture

Scheduler The Java program `dist.Scheduler` coordinates the node interaction. The scheduler process attached to the frontend kernel starts instances of the scheduler on other machines and communicates with them via sockets.

Maple Interface The program `dist.maple` running on every Maple kernel implements the interface between kernel and scheduler. Both components use pipes to exchange messages (which may embed any Maple objects).

The user interacts with Distributed Maple via the Maple frontend by a number of programming commands, in particular:

`dist[start](`*`f, a, ...`*`)` creates a task evaluating *f(a, ...)* and returns a task reference *t*. Tasks may create other tasks and arbitrary Maple objects (including task references) may be passed as arguments and returned as results.

`dist[wait](`*`t`*`)` blocks the execution of the current task until the task represented by *t* has terminated and returns its result. Multiple tasks may independently wait for and retrieve the result of the same task *t*.

This parallel programming model is based on para-functional principles which is sufficient for many kinds of computer algebra algorithms. In addition, the environment supports a non-deterministic form of task synchronization for speculative parallelism and shared data objects which allow tasks to communicate by side effects on a global store.

4 Experimental Results

We have benchmarked the parallel critical point computation in three environments with four random algebraic curves for which the sequential program takes 6870s, 470s, 155s, and 11748s (measured on a PIII@450MHz PC):

- The cluster of our institute which comprises 4 Silicon Graphics Octanes (2 R10000@250Mhz each) and 16 Linux PCs (various Pentium processors),
- a Silicon Graphics Origin multiprocessor (64 R12000@300 Mhz, 24 processors used) located at the university campus,
- a mixed configuration consisting of our 4 dual-processor Octanes and 16 processors of the Origin multiprocessor connected via an ATM line.

The raw computing power of each environment is 18.3, 17.1, and 18.7 (multiplication factors compared with a single PIII@450MHz processor); the benchmark results are as follows:

Execution Times (s)							
Example	Environment	1	2	4	8	16	24
1 (6870s)	Cluster	14992	8035	2732	1186	552	488
	Origin	7290	4217	1789	872	446	513
	Mixed	14992	8035	2732	1368	597	519
2 (470s)	Cluster	810	648	328	173	95	108
	Origin	667	541	297	166	112	116
	Mixed	810	648	328	210	116	127
3 (155s)	Cluster	267	191	112	67	46	45
	Origin	196	147	90	63	54	54
	Mixed	267	191	112	74	58	56
4 (11748s)	Cluster	25178	15559	6820	3562	1915	1563
	Origin	13397	8223	4009	2281	1726	1420
	Mixed	25178	15559	6820	4042	2004	1599

A detailed analysis of the results and our conclusions are given in a longer version of this paper at `http://www.risc.uni-linz.ac.at/software/distmaple`.

References

[1] C. Mittermaier. Parallel Algorithms in Constructive Algebraic Geometry. Master's thesis, Johannes Kepler University, Linz, Austria, 2000.

[2] M. Mnuk and F. Winkler. CASA - A System for Computer Aided Constructive Algebraic Geometry. In *DISCO'96 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 1128 of *LNCS*, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.

[3] T. Q. Nam. Extended Newton's Method for Finding the Roots of an Arbitrary Systemof Equations and its Applications. In *IASTED'94: 12th International Conference on Applied Informatics*, Annecey, France, 1994.

[4] W. Schreiner. Developing a Distributed System for Algebraic Geometry. In B. H. Topping, editor, *EURO-CM-PAR'99 Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics*, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press, Edinburgh.

PCI-DDC Application Programming Interface: Performance in User-Level Messaging

Eric Renault, Pierre David, and Paul Feautrier

Laboratoire PR*i*SM
Université de Versailles - Saint-Quentin-en-Yvelines
78035 Versailles Cedex, France
{Eric.Renault, Pierre.David, Paul.Feautrier}@prism.uvsq.fr

Abstract. Many programming interfaces which deal with peripherals (especially network peripherals) need to access the critical resources of the operating system, and system calls (or drivers) are generally used. Unfortunately, the time spent on such system calls is often greater than those requested for the operations themselves. In the case of the MPC machine, most of these operations do not need an intervention from the system and resources may be available from user applications. Our programming interface provides different levels of integration in the kernel depending on the security and the performance expected by the administrator. This article presents PAPI user-level performance.

1 Introduction

The MPC project started in 1995 as a collaborative endeavour between the LIP6 and PR*i*SM laboratories (France). Its goal is the development of both hardware and software layers to use the new HSL technology, a high speed network with an affordable price. An MPC machine is composed of nodes, each one with one or more processors and 1-Gbits/s links (HSL links). On each node, a Fast-HSL card is the interface between the network and the computer. An HSL link (IEEE 1355 [1]) is a bidirectional serial link which delivers a throughput of 1 Gbit/s. As 4 bits of control (generated by the crossbar to perform various controls) are transmitted with every byte, the effective bandwidth is limited to 80 MBytes/s. The RCube (for Rapid Reconfigurable Router) component [2] is a 8×8 crossbar with a high bandwidth of 640 MBytes/s and a low latency of 150 ns per hop. Routing in this chip is wormhole and adaptative and it is possible to define different configuration tables for each of the 8 bidirectional links. The PCI-DDC (for PCI-Direct Deposit Component) component [3] is a network interface for message-passing multiprocessor architectures using HSL links and RCube routers. It is connected to the PCI bus (where it can act as a master) and sends/receives messages to/from the RCube component without processor intervention, using Direct Memory Access. A Fast-HSL card is a 32-bit 33-MHz PCI card which includes the PCI-DDC and RCube components. It provides seven HSL ports, each one connected to a RCube entry; the last entry of the RCube chip is connected to the PCI-DDC component. In this article, we

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1201–1205, 2000.
© Springer-Verlag Berlin Heidelberg 2000

present a short description of the PCI-DDC component and a fast presentation of PAPI. Then, our user-level results are compared to those of Active Messages, BIP and Fast Messages.

2 Programming the PCI-DDC Component

The aim of the PCI-DDC component is to exchange messages with other nodes via the HSL network and the RCube components. Interface with the CPU uses two circular lists in main memory and some registers to store pointers in these lists. The two lists are the LPE (List of Pages to Emit) where CPU describes messages to send and the LMI (List of Message Identifiers) where the PCI-DDC component writes the description of each received message. A message identifier is a 24-bit integer which identifies a message; a single node does not permit two incoming messages with the same identifier at the same time.

The component allows the application to send two different message types: normal messages are using the "remote-write" protocol [4], i.e. both local and remote physical addresses must be specified by the sender; short messages do not need any physical address, i.e. instead of specifying local and remote physical addresses, the content of the message is placed directly in these fields. Short messages are particularly important during the initialization phase of an application, for example, since they are the only way to exchange physical addresses needed for normal messages. The size of these short messages is limited to 8 bytes, the room available in LPE and LMI entries.

PAPI [5], which stands for PCI-DDC Application Programming Interface, is a modular interface. It allows the system administrator to choose the security level of the system depending on the performance desired and the users. In the current version, three levels of integration are available:

- `NONE`: no security is provided. All of the module code is placed in user mode except some system calls (used only at initialization), used to access information from the kernel;
- `ACCESS`: traditional system security is provided, i.e. code is placed in kernel space and a system call is performed each time the application calls functions from our API;
- `HIDE`: the total security configuration, i.e. the code is placed in kernel space and no system call is provided to access it. The only way to run a function of a hidden module is to have another module with an `ACCESS` configuration, and system calls to access the functions of the hidden module.

3 Performance

The program used to determine the performance times of the MPC machine (see below) is the classical ping-pong program. It sends and receives one million messages between two nodes (one message can be sent if the message from the remote node has been received). Once this is done, the elapsed time is divided

by two millions in order to determine the total average time to send one message from one node to another one (i.e. one-way latency). The effective user bandwidth is computed by dividing the size of messages by the latency (throughout this section, we will refer to the effective user bandwidth as bandwidth).

At the PR*i*SM laboratory, each node of our MPC machine is composed of two 233-MHz Pentium II with 32 MB of memory. Another network (a 10-Mbit/s Ethernet TCP/IP network) is used for control operations, such as program launching. The operating system is FreeBSD 3.2. We will discuss our results by comparing them with Myrinet [6] message-passing libraries, especially Active Messages [7], BIP [8] and Fast Messages [9]. These support an architecture similar to that of the MPC project, i.e. a cluster of PCs, and are some of the fastest available. Performance results from these other message-passing libraries are taken from the literature [10][11][12], and not measured directly.

Figure 1 shows the time required to send a message from one node to another for different message-passing libraries. It shows that a latency for message sizes less than or equal to 64 bytes is quite similar for BIP, Fast Messages and PAPI. However, PAPI has a far better latency for messages larger than 64 bytes.

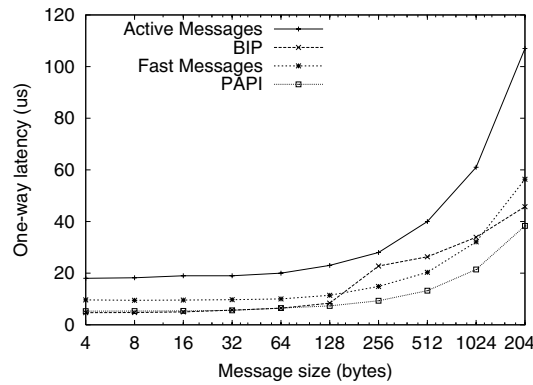


Fig. 1. One-way latency for different message-passing libraries

The anomaly in the BIP curve above 128 bytes (see Fig. 2) is due to the short message concept of BIP. The boundary is software-dependent, and reflects the threshold where messages are sent directly to the host memory rather than copied in the adapter memory. The PCI-DDC concept of short message is quite different: short message data are copied in the LMI of the receiving node, and are restricted to 8 bytes, the room available in LPE and LMI entries.

Where is the latency time spent? Figure 3 analyses the details of the transmission of a short message. On the sender node, the application calls our `PAPI_SSend` function, which fills an entry in the LPE in memory, performs a write in a PCI-DDC register which is located in PCI configuration space, and returns. Once

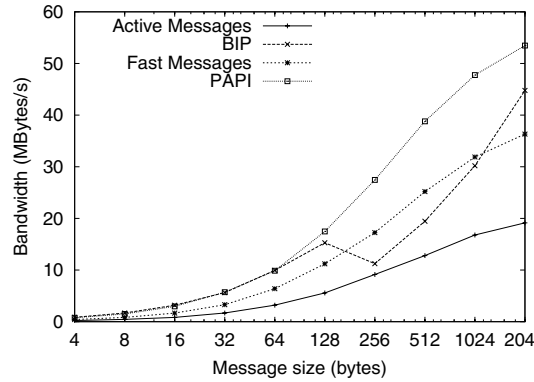


Fig. 2. Bandwidth for different message-passing libraries

informed by the write that a new message must be sent, the PCI-DDC component starts to work. Hardware specifications [3][2] tell us that the components and the network spend 1.7 μs (hardware latency) to bring the message to the other node memory. On the receiver node, where the `PAPI_Receive` function is called by the application, the LMI list is monitored. As soon as it is updated by the receiving PCI-DDC component, `PAPI_Receive` returns with information about the incoming message.

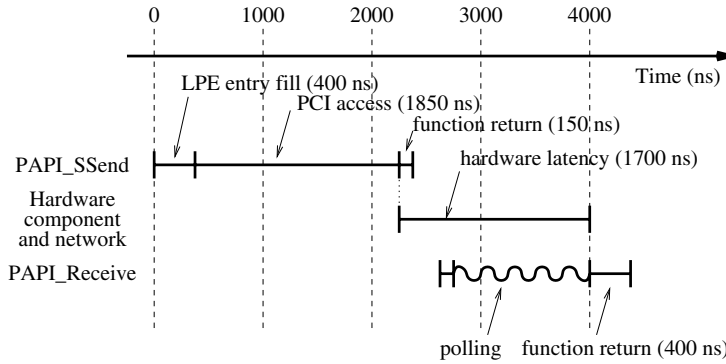


Fig. 3. Decomposition of the transmission time for a short message

This figure highlights another important point: during the transmission of the message, only a little part (0.8 μs over 4.39 μs, i.e. less than 20%) depends upon the CPU speed. The rest of the time depends upon the PCI bus speed and the hardware latency. The greatest part (more than 80%) of the latency time is

thus independent of the CPU. So this means that PAPI is close to the minimum latency for the Fast-HSL card.

4 Conclusion

In this article we have presented PAPI, an application programming interface for the MPC machine. We have shown that, even for messages of less than one system page (4 kB), our programming interface takes advantage of the HSL link bandwidth. The latency is very close to the minimum latency one can expect from the Fast-HSL cards.

Our study shows that a large part of the latency time is spent on a PCI access through a configuration space register. Given the same HSL network and router (RCube), improvements could be made only with modified access to the PCI-DDC registers, through a memory-mapped access. However, this would require a modification of the PCI-DDC hardware component.

References

[1] C. Whitby-Strevens and al. IEEE Draft Std P1355 — Standard for Heterogeneous Interconnect — Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction, 1993.

[2] V. Reibaldi. *RCube Specifications*. Laboratoire Informatique de Paris VI, February 1997.

[3] J.J. Lecler, F. Potter, A. Greiner, J.L. Desbarbieux, and F. Wajsburt. *PCI-DDC Specifications*. Laboratoire Informatique de Paris VI, 1996.

[4] F. Potter. *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs*. PhD thesis, Université Pierre et Marie Curie, 1996.

[5] E. Renault. Pci-ddc Application Programming Interface User Manual. Research report, Laboratoire PR*i*SM, Université de Versailles – Saint-Quentin, France, May 2000.

[6] N.J. Boden, D. Cohen, R.E. Flederman, A.E. Kulawik, C.L. Seitz, J.N. Selzovic, and W.-K. Su. Myrinet – A Gigabit-per-Second Local-Area Network. In *IEEE-Micro*, volume 15, pages 29–36, 1995.

[7] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Mesages: a Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, 1992.

[8] L. Prylli. *BIP Messages User Manual for BIP 0.94*, June 1998.

[9] S. Pakin, V. Karamcheti, and A.A. Chien. Fast Message (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. In *IEEE Concurrency*, volume 5, pages 60–63, 1997.

[10] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-Space Communication: A Quantitative Study. In *The SuperComputing*, Orlando, Florida, 1998.

[11] Concurrent Systems Architecture Group. Fast Messages Performance. Web page, 1999. http://www-csag.ucsd.edu/projects/comm/fm-perf.html.

[12] High Speed Networks and Multimedia Application Support. BIP Performance Curves. Web page, 1997. http://lhpca.univ-lyon1.fr/Resultats/bipres.html.

A Clustering Approach for Improving Network Performance in Heterogeneous Systems*

Vicente Arnau, Juan M. Orduña, Salvador Moreno, Rodrigo Valero, and Aurelio Ruiz

Departamento de Informática. Universidad de Valencia.SPAIN
Juan.Orduna@uv.es

Abstract. A lot of research has focused on solving the problem of computation-aware task scheduling on heterogeneous systems. In this paper, we propose a clustering algorithm that, given a network topology, provides a network partition adapted to the communication requirements of the applications running on the machine. Also, we propose a criterion to measure the quality of each one of the possible mappings of processes to processors based on that network partition. Evaluation results show that these proposals can greatly improve network performance, providing a basis of a communication-aware scheduling technique.

1 Introduction

Networks of Workstations (NOW's) are used nowadays as parallel computers, forming heterogeneous systems. A lot of research has focused on solving the *NP-complete* problem of efficiently scheduling diverse groups of tasks to the machines that form these systems from the computational point of view. However, the increasing computational power of new processors and the growing communication requirements of the applications may cause the interconnection network in these heterogeneous systems to become the system performance bottleneck.

In a previous paper, we proposed a new model of communication cost between nodes, *the table of equivalent distances* [1]. This model consist of a table of $N \times N$ distances, where N is the number of nodes in the network. In this table, the element T_{ij} represents the communication cost for messages going from node i to node j. In this paper, we propose a clustering algorithm based on the table of distances that provides a network partition, and a criterion to measure the suitability of each allocation of network resources to the applications that the provided network partition may generate. Evaluation results show that the use of this proposals significantly improve network performance by reducing communication bottlenecks. Furthermore, this clustering technique is applicable to both regular and irregular topologies, providing a general basis for communication-based process mapping.

2 A New Clustering Approach

From a general point of view, we can consider that each application belongs to a different user. Therefore, we can assume that the processes belonging to the same application

* Supported by the Spanish CICYT under Grant TIC97–0897–C04–01

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1206–1209, 2000.
© Springer-Verlag Berlin Heidelberg 2000

may intensively communicate between them, but they will not communicate at all with processes from other applications. Therefore, we can group the processes running on the machine, forming a set of logical clusters of processes, where each cluster is formed by the processes belonging to each application. The proposed algorithm intends to provide a network partition adapted to any existing set of logical clusters.

The first step in this method is to find an Euclidean metric space in which we can represent our N nodes, in such a way that the resulting distances between them are as close as possible to the values in the table of distances (the latter one does not define a metric space). We have computed a least squares linear adjustment using the steepest gradient method [2]. The solution consists of N points in an Euclidean space whose coordinates generate a table of Euclidean distances with the least quadratic error with regard to the table of distances. It is worth mention that the table of Euclidean distances does not contain repeated values (except zero values in the diagonal).

Once a table of Euclidean distances has been computed, the furthest-neighbor algorithm is used to compute the optimal dendogram [4]. This algorithm uses a similarity measure. In each step the algorithm merges two of the existing clusters into a new one, choosing the two clusters that result in the lowest similarity measure when the step is applied. The similarity measure usually used in this algorithm is the intracluster distance, and therefore it is called the furthest-neighbor algorithm. However, we considered as the similarity measures f to be *maximized* the inverse of the Euclidean distance, defined as $f_a = \frac{1}{D_{ij}}$, where D_{ij} is the distance from cluster i to cluster j in the Euclidean table of distances. The initial network partition consists of the N nodes located at the coordinates given by the computed table of Euclidean distances. In each step a new partition is formed, decreasing the number of clusters by one. When merging two clusters, they are replaced by a new cluster, and the Euclidean table of distances must be computed again in each step.

The result of the above clustering approach is a dendogram, but not a mapping of processes to processors. The cardinal of the set of logical clusters can be used to determine when to stop the clustering algorithm, obtaining a network partition with a number of network clusters equal to the number of existing logical clusters of processes. Nevertheless, the number of nodes (switches) in each network cluster may significantly differ from the number of processes in each logical cluster of processes. Therefore, new changes in this partition are still needed in order to map all the existing processes according to the communication requirements. We have performed manually this clustering adjustment, obtaining different possible process mappings from each network partition. However, it is necessary to define a metric of the communication bandwidth achieved by each one of the possible mappings, in order to select the one who provides the best network performance.

We have defined two distinct and complementary global quality functions, the *similarity* function F_G and the *dissimilarity* function D_G. F_G measures the intracluster distances, and D_G measures the intercluster distances. The cluster quality function F_{A_i} for cluster A_i is defined as the quadratic sum of all intracluster distances. It must be noticed that for these functions we are considering the distances in the table of distances. Although the partition provided by the Euclidean approach is based on an Euclidean table of distances, the quality function must be based on the table of distances, since it mea-

sures the actual network distances. The similarity global function for the final partition F_G is computed as the sum of all the F_{A_i} values divided by the total number of intracluster distances existing in partition P and normalized by the quadratic average value of all of the distances between the network nodes. For the dissimilarity global function we define the cluster dissimilarity function D_{A_i} for a cluster A_i as the quadratic sum of all intercluster distances from nodes in cluster A_i to all the nodes in the rest of the clusters. The dissimilarity global function D_G is defined as the sum of all the D_{A_i} values divided by the total number of existing intercluster distances in partition P and normalized by the quadratic average value of all of the distances between the network nodes. F_G and D_G provide a measurement of the intracluster and intercluster communication costs, respectively. Thus, the quotient $D_G/\ F_G$ provides the relationship between the intracluster and intercluster bandwidth achieved with each process mapping. We will denote this relationship as the *clustering coef ci ent* C_c. This clustering coefficient can be used to measure the quality of each process mapping.

3 Performance Evaluation

We have evaluated the improvement in network performance that the proposed clustering approach can provide, as well as the correlation between the clustering coefficient and network performance. This study assumes that all the communication between processors is intracluster communication and all the processors transmit the same amount of information. We have evaluated the performance of several irregular networks by simulation. The evaluation methodology used is based on the one proposed in [3]. The most important performance measures are latency and throughput.

The network is composed of a set of switches. The network topology is irregular and has been generated randomly. We assumed 8-port switches. Each switch has 4 ports available to connect to other switches. From these 4 ports, three of them are used in each switch when the topology is generated. The remaining port is left open. We have evaluated networks with a size ranging from 16 switches (64 nodes) to 24 switches (96 nodes). Several distinct topologies have been analyzed. For the sake of simplicity, we have assumed a fixed pool of N processes grouped into 4 clusters with $\frac{N}{4}$ processes, where N is the number of network nodes. Each process is assumed to send all of the generated messages to processes in the same logical cluster of processes. For each network, we have computed the clustering algorithm until it has provided a 4-cluster partition of the network, and then we have chosen several possible mappings based on this partition. Additionally, we have computed several random mappings for each considered network.

Figure 1 shows network performance for some of the mappings based on the partition provided by the Euclidean approach (E_i labels) for a 16–switch network, compared with the network performance obtained by several randomly generated mappings (R_i labels). The clustering coefficient C_c obtained for each mapping is shown on the right side of each plot label. The network throughput obtained with any of the mappings based on the proposed approach is about a 55% higher than the network throughput obtained with any of the randomly generated mappings, while the network latency is less

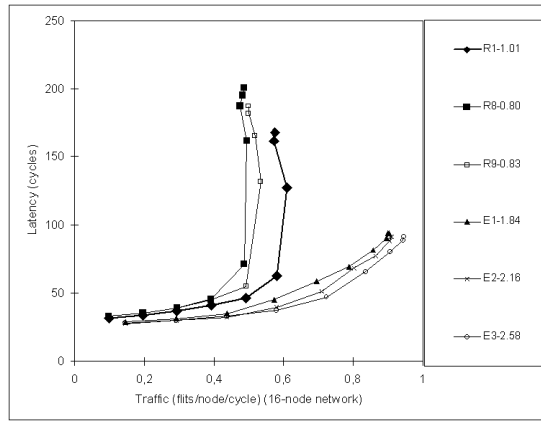


Fig. 1. Simulation results for a 16-switch network

than 63%. On the other hand, the value of C_c is clearly lower for randomly generated mappings, showing that this function is directly related to network performance.

We have also studied the correlation of the clustering coefficient C_c with network performance. We computed the correlation index between the clustering coefficient and the network performance obtained for all the mappings in all of the considered networks. In any case this index resulted higher than 80% for simulation points of both low network load and network saturation. These results validates the clustering coefficient as an "a priori" measure of relative network performance.

4 Conclusions

Network throughput and network latency are greatly improved when using the mappings based on the proposed approach, showing that it can be used as the basis for a communication-aware mapping technique. We have also studied the correlation between the proposed clustering coefficient and network performance. The results show that when only exists intracluster communication then this coefficient is highly correlated with network performance. For further information, please see technical report TR-AR99 on *http://www.gap.upv.es*

References

1. V. Arnau et al., "On the Characterization of Interconnection Networks with Irregular Topology: A New Model of Communication Cost", in *PDCS 99*, November 1999.
2. M. S. Bazaraa et al., *Nonlinear Programming: Theory and Algorithms*, *J. Wiley, 1993*.
3. J. Duato, "A new theory of deadlock-free adaptive routing in wormhole networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, December 1993.
4. R. Duda, P. Hart, *Pattern Classi cat ion and Scene Analysis*, *J. Wiley, 1973*.

Topic 19
Metacomputing

Alexander Reinefeld, Geoffrey Fox, Domenico Laforenza, and Edward Seidel

Topic Chairmen

The basic idea about metacomputing is to utilize a variety of geographically dispersed resources, such as computers, storage systems, data sources and special devices, which are seen by the user as a single unified resource. This new computing paradigm was born with the growing success of the Internet which made it possible to link remote high-performance computers for the collaborative use.

A metacomputer provides a variety of capabilities that can be orchestrated to execute multiple tasks with varied computational requirements. Applications in these environments achieve their performance by properly mapping the tasks onto the best suited platforms while considering the overhead of inter-task communication and the coordination of distinct data sources and administrative domains.

Ideally, the distributed nature of a metacomputer environment is transparent to the user, that is, the user only needs to describe the constraints connected with the job while the system selects the most suitable machine for the execution of his job. This selection process is subject to a large variety of different constraints including access restrictions, user priorities, machine workload, job characteristics and user preferences. In addition the metacomputer structure may change due to maintenance shut downs or temporary failures of sub-systems. It is the task of the management software to handle those problems, that is, to provide a transparent access to the users while considering any special requests from users and owners. This metacomputer management software is the core topic of the research described in the following papers.

The paper by Arnold, Bachmann, and Dongarra presents a general technique to reduce network traffic when executing several requests to a grid computing system. This technique, called "request sequencing", uses DAGs for building task sequences which allows to minimize the data transfer by grouping requests. Request sequencing can thereby affect scheduling policies and enable more expedient resource allocation methods.

The paper by Kindermann and Fink describes an approach for a component-based meta-applications design based on formal architecture description of the gross organization of an application. Simple architectural styles developed to support data-flow and control-flow driven meta-application design on top of the *Amica* metacomputing infrastructure are presented.

The paper by Kamachi, Priol and René describes the use of distributed objects (i.e., parallel CORBA objects) as a modern approach for programming computational grids. In particular, the authors focus on the problems related to

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1211–1212, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the handling of distributed data within parallel objects, showing some interesting performance results on a practical distributed computing platform.

The paper by Neary, Phipps, Richman, and Cappello focuses on Java-based parallel computing in the Internet. It describes enhancements to the well-known *Javelin* system. Javelin aims at freeing the application developer from concerns about processor interconnect issues, thereby allowing to focus on the application issues.

Request Sequencing: Optimizing Communication for the Grid*

Dorian C. Arnold[1], Dieter Bachmann[2], and Jack Dongarra[1]

[1] Department of Computer Science, University of Tennessee, Knoxville, TN 37996
[darnold, dongarra]@cs.utk.edu
[2] Computer Graphics and Vision, Graz University of Technology,
Inffeldg. 16/E/2, A-8010 Graz Austria
bachmann@icg.tu-graz.ac.at

Abstract. As we research to make the use of Computational Grids seamless, the allocation of resources in these dynamic environments is proving to be very unwieldy. In this paper, we introduce, describe and evaluate a technique we call request sequencing. Request sequencing groups together requests for Grid services to exploit some common characteristics of these requests and minimize network traffic. The purpose of this work is to develop and validate this approach. We show how request sequencing can affect scheduling policies and enable more expedient resource allocation methods. We also discuss some of the reasons for our design, offer the initial results and discuss issues that remain outstanding for future research.

1 Introduction

The vBNS[1] and Myrinet[2] represent two technologies that connect computational resources at high speeds in settings from local to global area networks. However, with the speed of processors increasing at a rate much greater than that of networking infrastructure, data transfer continues to bear large overhead for many applications of high-performance computing. Yet, straightforward ways for increasing application performance by optimizing communication often go overlooked. Our research on call sequencing for Grid middleware aims to take a significant step in this direction.

Computer applications generally exhibit two common characteristics: large input data sets and data dependency amongst computational cycles. The goal of this effort is to employ simple, yet highly effective, strategies for exploiting these characteristics. We believe that not enough attention has been given to examining ways to effectively distribute application data amongst the different components of a Grid. We qualify this last statement by saying that data partitioning has been well researched for parallel programming, but in cases where

* This work supported in part by Raytheon Systems subcontract #AA23, PACI subaward #790 under prime NSF Cooperative Agreement #ACI-9619019 and NSF Grant #ACI-9876895.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1213–1222, 2000.
© Springer-Verlag Berlin Heidelberg 2000

computational modules execute concurrently with no data exchange, often the same data is unnecessarily transported multiple times between the same components.

1.1 Positioning Our Work

This paper explores the design, implementation and initial results of what we call request sequencing. This term encompasses both an interface to group a series of requests and a scheduling technique, viz. one that uses data persistence and a Direct Acyclic Graph (DAG) representation of computational modules. Our motivation was to allow users to take advantage of data redundancies within a sequence of requests and optimize data communications. We developed and tested our ideas using the NetSolve system described in Sect. 2. Below, we offer the relationship of this research to other research that has been or is being done.

Our scheduling work is reminiscent of techniques utilized in schedulers that execute "batches" of processes on parallel machines. We create task graphs or DAGs that represent execution dependencies and schedule them for execution [3]. J. Dennis researched data flow scheduling techniques for Supercomputers[4]; our work presents a similar idea for Grid environments.

Ninf [5] is a functional metacomputing environment that shares many similarities with NetSolve. The project has implemented a strategy to increase parallelism amongst the computational services. Similar to this work, they group together requests and execute the modules simultaneously, when possible. Their main focus is on parallel module execution, and it is not stated whether redundant messages are sent or not. Our focus is on minimizing network traffic; our design ensures that no unnecessary data transfer takes place.

Condor [6] is a high-throughput computing system that manages very large collections of distributively owned workstations. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor jobs. Users can submit batch jobs to the Condor system and use DAGMan to pre-define execution order. Once again, however, the main focus is on parallel execution and not data transfer. As an extra burden, the data dependency analysis is left to the user.

The contribution put forth by this paper is a thorough understanding of an approach to optimize data transfer in Grid settings and the empirical data to justify using this approach. We also offer discussion of scheduling in this environment. Section 2 of this paper presents details about NetSolve, which is our deployment environment. Section 3 describes the design and implementation of the sequencing interface, the server data persistence and execution scheduling. Section 4 contains the experimental test cases and the results that validate this strategy. Finally, Sect. 5 summarizes the work and discusses future research goals.

2 An Overview of NetSolve

The NetSolve [7] project is being developed at the University of Tennessee. It provides remote access to computational resources, both hardware and software.

Built upon standard Internet protocols, like TCP/IP sockets, it supports popular variants of UNIX, and the client is available for Microsoft Windows '95, '98, NT and '00.

Figure 1 shows the infrastructure of the NetSolve system and its relation to the applications that use it. NetSolve and similar systems are referred to as Grid Middleware; this figure explains this terminology. The shaded areas represent the NetSolve system; it can be seen that NetSolve acts as a glue layer that brings the application or user together with the hardware and/or software it needs to complete useful tasks.

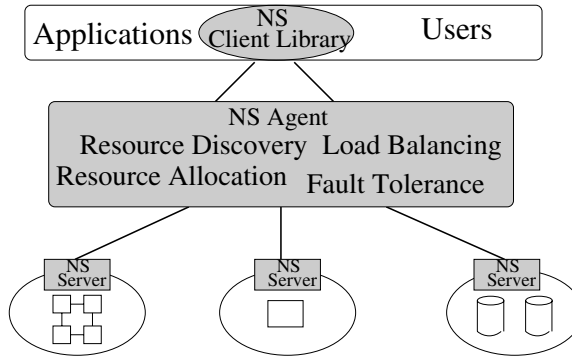


Fig. 1. Architectural Overview of the NetSolve System

At the top tier, the NetSolve client library is linked in with the user's application. The application then makes calls to NetSolve's application programming interface (API) for specific services. Through the API, NetSolve client-users gain access to aggregated resources without the users needing to know anything about computer networking or distributed computing.

The NetSolve agent maintains a database of NetSolve servers along with their capabilities (hardware performance and allocated software) and dynamic usage statistics. It uses this information to allocate server resources for client requests. The agent finds servers that will service requests the quickest, balances the load amongst its servers and keeps track of failed ones.

The NetSolve server is a daemon process that awaits client requests. The server can run on single workstations, clusters of workstations, symmetric multiprocessors or machines with massively parallel processors. A key component of the NetSolve server is a source code generator which parses a NetSolve problem description file (PDF). This PDF contains information that allows the NetSolve system to create new modules and incorporate new functionalities.

3 Sequencing Design and Implementation

As stated in Sect. 1, our aim in request sequencing is to decrease network traffic and overall request response time. Our design needs to ensure that i) no unnecessary data is transmitted and ii) all necessary data is transferred. We also need to cut execution time by executing modules simultaneously when possible. We do this by performing a detailed analysis of the input and output parameters of every request in the sequence to produce a DAG that represents the tasks and their execution dependencies. This DAG is then sent to a server in the system where it is scheduled for execution.

3.1 The DAG Model

Kwok et al. [3] offers a very good description of the DAG:

> The DAG is a generic model of a parallel program consisting of a set of processes (nodes) among which there are dependencies. A node in the DAG represents a task which in turn is a set of instructions that must be executed sequentially without preemption on the same processor. A node has one or more inputs. When all inputs are available, the node is triggered to execute. The graph also has directed edges representing a partial order among the tasks. The partial order introduces a precedence-constrained directed acyclic graph and implies that if $n_i \rightarrow n_j$, then n_j is a child which cannot start until its parent n_i finishes and sends its data to n_j.

3.2 Data Analysis and the DAG

In order to build the DAG or task graph, we need to analyze every input and output in the sequence of requests. We evaluate two parameters as the same if they share the same reference. We use the size fields and reference pointer of the input parameters to calculate when inputs overlap in the memory space. NetSolve supports many object[1] types, including matrices, vectors and scalars; only matrices and vectors are checked for reoccurences on the premise that these are the only objects that tend to be large enough for the overhead of the analysis to pay dividends. This analysis yields a DAG. The graph is acyclic because looping control structures are not allowed within the sequence, and therefore, a node can never be its own descendant.

3.3 The Interface

In addition to the original function used for request submittal, two functions are implemented; their purpose is to mark the beginning and end of a sequence of

[1] We use the term object to refer to a composition of native data types, as in a matrix object of native integers

requests. `begin_sequence()` takes no arguments and returns nothing; it notifies the system to begin the data analysis. `end_sequence()` marks the end of the sequence; at this point, the sequence of collected requests is sent to a server(s) to be scheduled for execution. As an enhancement, this function also takes a variable number of arguments describing which output parameters NOT to return. This means that if the intermediate results are not necessary for any local computations, they need not be returned. This is a part of the API as it is the user who should determine which results are mandatory and which are useless. Figure 2 illustrates what a sequencing call might look like. Two points to note in this example: i)for all requests, only the last parameter is an output, and ii)the user is instructing the system not to return the intermediate results of `command1` and `command2`.

```
...
begin_sequence();
submit_request("command1", A, B, C);
submit_request("command2", A, C, D);
submit_request("command3", D, E, F);
end_sequence(C, D);
...
```

Fig. 2. Sample C Code Using Request Sequencing Constructs

For the system to be well-behaved, we must impose certain software restrictions upon the user. Our first restriction is that no control structure that may change the execution path is allowed within a sequence. We impose this restriction because the conditional clause of this control structure may be dependent upon the result of a prior request in the sequence, and since the requests are not scheduled for execution until the end of the sequence, the results will likely not be what the programmer expects.

The other restriction is that statements that would change the value of any input parameter of any component of the sequence are forbidden within the sequence (with the exception of calls to the API itself that the system can track.) This is because during the data analysis, only references to the data are stored. So if changed, the data transferred at the end of the sequence will not be the same as the data that was present when the request was originally made. We contemplated saving the entire data, rather than just the references, but this directly conflicts with one of our premises – that the data sets are large; multiple copies of these data are not desirable.

3.4 Execution Scheduling at the Server

Once the entire DAG is constructed, it is transferred to a NetSolve computational server. [3] offers a taxonomy of graph scheduling algorithms in multi-processor

environments. These algorithms take into account both node-computation and inter-node communication costs. In this first version of request sequencing, the NetSolve agent uses a larger granularity and decides which server should execute the entire sequence. We execute a node if all its inputs are available and there are no conflicts with its output parameters. The reason for this is that currently the only mode of execution we support is on a single NetSolve server – though, that server may be a symmetric multi-processor (SMP). We discuss our plans for expanding this model in Sect. 5.

For data partitioning, we transfer the union of the input parameter sets to the selected server host. This makes input for all nodes, except those which are intermediate output from prior nodes, available for the execution of the sequence. When we move to a multi-server execution mode for the sequence, we must enhance our data staging technique, and this is also discussed in Sect. 5.

Since all execution is on a single server, our execution scheduling algorithm is reasonably uncomplicated. In essence, we execute all nodes with no dependencies, updating the dependency list as nodes complete, and then check for further nodes to execute. We do this until all nodes have executed:

```
while(problems left to execute)
  execute all problems with no dependencies;
  wait for at least one problem to finish;
  update dependencies;
}
```

3.5 Discussion

Figures 3 and 4 show the reduced network activity between client and server during execution of the sequence in Fig. 2. In the first case, input `A` is sent to the server twice, and output `C` and `D` are unnecessarily sent back to the client as intermediate output and then to the server once again as input. In the latter case, these unnecessary transfers are removed. (These diagrams show three potentially different servers, but our current implementation sees this as three instances of the same server.) Our hypothesis is that this reduction in data traffic will yield enough performance improvements to make sequencing worthwhile.

4 Applications and Initial Results

In this section, we discuss the applications that we used to test our request sequencing infrastructure. They are from the remote sensing/image processing domain as it was the nature of some of these applications that led us to investigate request sequencing. The size of images that are analyzed can become very large and easily extend into the gigabyte range. It is also common in many image processing applications to execute a series of operations on an image, usually one transformation after another.

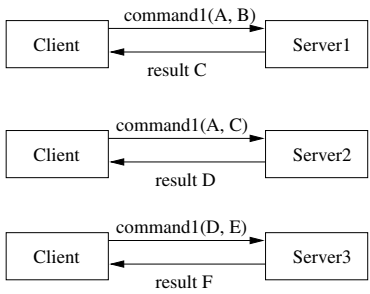


Fig. 3. Client-Server Data Flow Without Request Sequencing

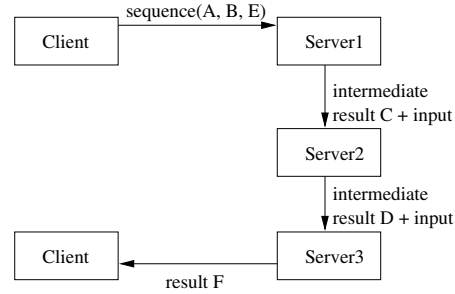


Fig. 4. Client-Server Data Flow With Request Sequencing

Experiments were executed from NetSolve clients connected to a switched 10/100Mbit ethernet and crossing a 155Mbit ATM switch that is directly connected to the NetSolve servers. The NetSolve server was an SGI Power Challenge with eight R10000 processors. Graphed results are the averages of four independent sets of runs.

For the experiments, we varied the network bandwidth by using the NistNet [8] interface on a Linux router. A network performance testing tool, TTCP, which is able to generate TCP traffic on IP based networks, was used to obtain a correction curve for the values set by NistNet.

4.1 Linear Sequence: Principle Component Analysis

Multispectral or multidimensional remote sensing data can be represented by constructing a vector space using one axis per dimension. By calculating the covariance matrix , the axes are transformed into an uncorrelated system. This transformation is called Principle Component Analysis (PCA) [9].

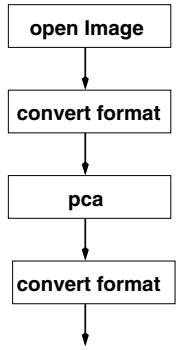


Fig. 5. Principal Component Analysis

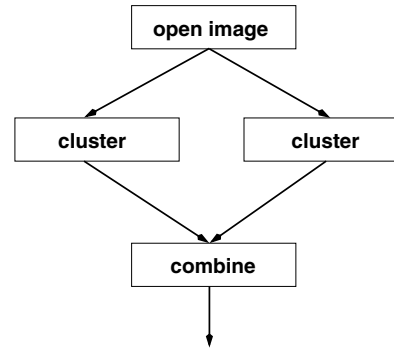


Fig. 6. Multimodal Image Clustering

In remote sensing, the PCA is used to reduce the number of channels for input images by moving the information towards the first bands. The application used for testing a linear sequence opens a 10MB image, performs a PCA and stores the transformed result. The structure is shown in Fig. 5.

Fig. 7 shows how the total response time (from request initiation to availability of results) varies with the bandwidth. It confirms our beliefs: with sequencing in place, there is a significant reduction in execution time of the PCA application. Similar results are to be expected for applications that exhibit similar levels of parameter sharing, and in fact, our examples are not contrived, but represent realistic applications that scientists have used to support their research.

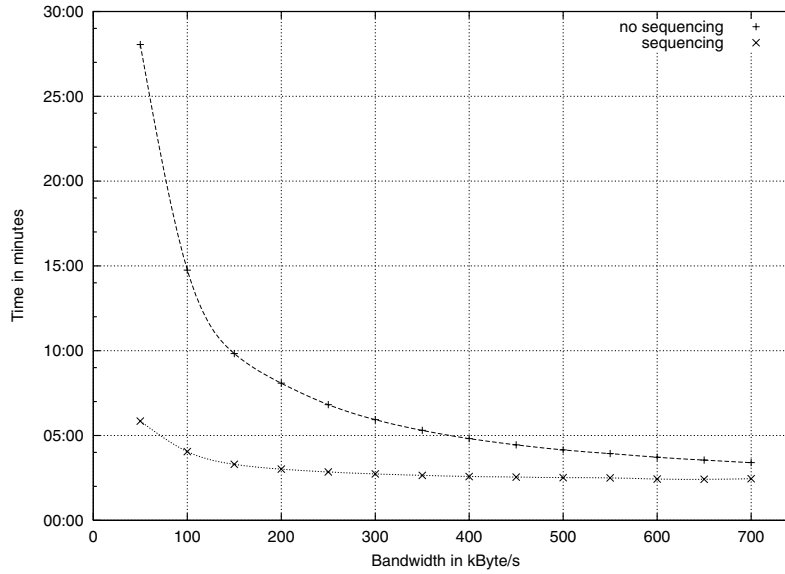


Fig. 7. PCA Sequence Executed on an SGI workstation

4.2 Parallel Sequence: Clustering

To handle multisource/multimodal satellite images or to improve clustering accuracy, several classification steps are performed, and their results are combined by a fusion module. Such a module can consist of either a simple pixel selection approach based on severity ratings or a knowledge based combination module [9]. For our tests a pixel based approach has been chosen using an image size on the order of 1MB. This process is illustrated in Fig. 6.

Fig. 8 graphs the variation of response time with bandwidth for this parallel sequence. The shape is similar to that of the PCA application. Again, request sequencing yields decreases in execution time.

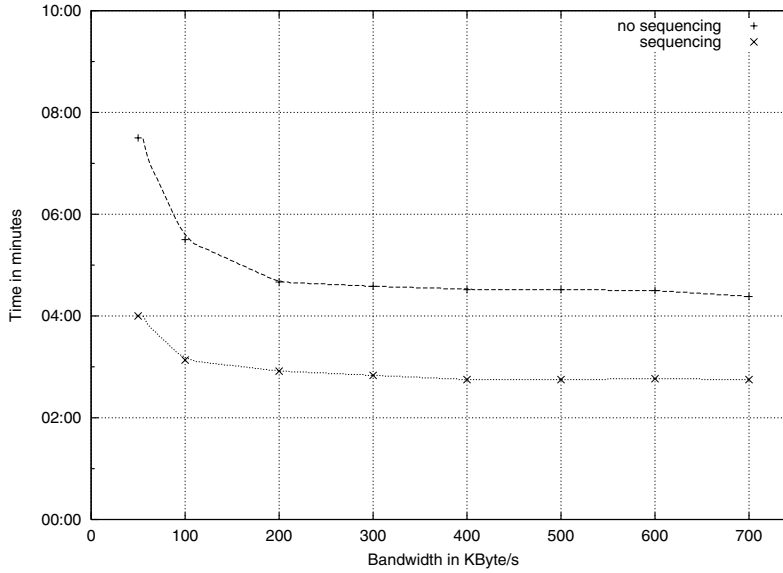


Fig. 8. Clustering Sequence Executed on Two Processors of an SGI workstation.

These preliminary results encourage our investigation of sequencing, hinting that a multi-server mode can be a very useful methodology in Grid Computing.

5 Conclusion and Future Work

We have presented a general technique to reduce network traffic when executing requests in Grid environments. Our approach is to build a DAG whose structure expresses the dependencies amongst the requests; the DAG is then scheduled for execution. Our initial experiments are promising and show that sequencing significantly reduces the execution time of our client application. As a final thought, we offer Fig. 9 which shows that even at its worst, request sequencing improves execution time by a factor of about 1.5. Though we have not proven this result, it is our belief that in most cases, sequencing should never decrease performance.

Section 3.5 mentions that the sequences are currently restricted to execution on a single server. The next logical progression is to allow different components of the sequence to execute on different hosts. The implications are that no single server needs to possess all the software capabilities for the sequence. This also means that the modules will truly be able to execute in parallel even when no parallel machine is present. Scheduling techniques as discussed by [3] will be evaluated, and we will incorporate factors like computational and communication costs to better approximate optimal solutions. It makes little sense to execute the components of a sequence on various servers without taking data locality into account. Tools like the Internet Backplane Protocol[10] will be leveraged to provide all servers with convenient access to the necessary data.

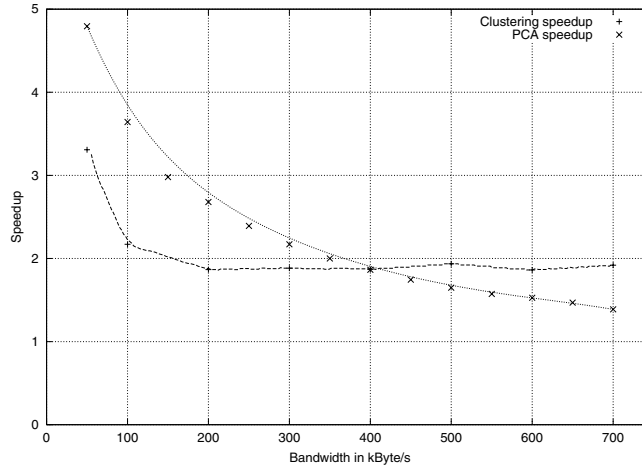


Fig. 9. Reduction in Execution Time due to Request Sequencing

References

[1] J. Jamison and R. Wilder. vBNS: The Internet Fast Lane for Research and Education. *IEEE Communications Magazine*, 35(1):60–63, January 1997.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit per Second Local Area Network. *IEEE-Micro*, 15:29–36, February 1995.

[3] Y. Kwok and I. Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, December 1999.

[4] J. Dennis. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[5] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf : Network based Information Library for Globally High Performance Computing. In *Proc. of Parallel Object-Oriented Methods and Applications (POOMA), Santa Fe, CA*, 1996.

[6] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th International Conference of Distributed Computing Systems, San Jose, CA*, pages 104–111, June 1988.

[7] H. Casanova and J. Dongarra. NetSolve's Network Enabled Server: Examples and Applications. *IEEE Computational Science & Engineering*, 5(3):57–67, September 1998.

[8] S. Parker and C. Schmechel. RFC 2398: Some testing tools for TCP implementors, August 1998.

[9] J. A. Richards. *Remote Sensing Digital Image Analysis.* Springer, 2nd edition, 1993.

[10] J. Plank, M. Beck, W. Elwasif, , T. Moore, M. Swany, and R. Wolski. IBP – The Internet Backplane Protocol: Storage in the Network. In *NetStore '99: Network Storage Symposium, Seatle, WA*, October 1999.

An Architectural Meta-application Model for Coarse Grained Metacomputing

Stephan Kindermann[1] and Torsten Fink[2]

[1] University of Erlangen-Nuremberg, Germany
snkinder@informatik.uni-erlangen.de
[2] Free University of Berlin, Germany
tnfink@computer.org

Abstract The emerging infrastructures supporting transparent use of heterogeneous distributed resources enable the design of a new class of applications. These meta-applications are composed of distributed software components. In this paper we describe a new model for component based meta-application design based on a formal architectural description of the gross organization of an application. This structural description is enriched by a formal process algebraic characterization of component behavior. Using this behavioral model we can formally check meta-applications in an early development phase. We present simple architectural styles developed to support data-flow and control-flow driven meta-application design on top of the Amica metacomputing infrastructure.

1 Introduction

There is a growing interest in defining and constructing an infrastructure which gives users the illusion that distributed, heterogeneous (computing and storage) resources constitute one giant transparent environment, a metacomputer [8]. It enables the development of a new class of applications: meta-applications. They are composed of multiple (partly reusable) components. Looking at the current practice of meta-application development there is no agreement on a common programming model for component based application design.

In this paper we describe an abstract, formally concise, and extendable programming model defined to develop meta-applications on top of our metacomputing environment Amica[1] [2]. The overall organization of meta-applications are given in a formal architecture description language (ADL) (see e.g. [6]). Different architectural styles [13] are defined to build up a basic description vocabulary which can be refined to define domain specific extensions. For every element of the vocabulary its behavior is defined by a process algebra term. The overall behavioral model of an application is automatically derived through the appropriate composition of the behavioral description of its components. This model can then be checked against a set of formal requirements (e.g. liveness

[1] Abstract Metacomputing Infrastructure for Coarse Grained Applications

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1223–1230, 2000.
© Springer-Verlag Berlin Heidelberg 2000

and progress properties). This allows formal correctness checks on Amica meta-applications in an early development phase.

The remainder of this article is as follows. In Sect. 2 we introduce shortly the basic concepts and services of Amica. In Sect. 3 we introduce our new programming model. In Sect. 4 we describe firstly how applications given in our programming model are executed using Amica and secondly how a behavioral model of the application is generated automatically which can be used as input to formal analysis tools. In Sect. 5 we give an overview of related work and discuss the advantages of our approach. Finally, an outlook to future work is given.

2 The Amica Metacomputing Infrastructure

Amica has been designed as a prototypical middleware foundation to investigate the composition of metacomputing applications from reusable components (e.g. legacy systems). Amica provides abstraction of heterogeneous distributed data storage by *data objects* and of computing facilities by *metabricks*. Additional application specific code can be integrated using so called *user bricks*. Data objects (possibly replicated) are related dynamically to real storage resources (data store objects) interconnected by link objects which provide abstraction of the networking infrastructure of the metacomputer. Metabricks are related to the basic computational services (bricks) based on a broker mechanism which looks for appropriate brick factories. The concise meaning of 'appropriate' is given by a cost function which takes into account the current load of computing resources provided by special objects named computation units. Essentially, Amica provides an infrastructure to carry out the instantiation of abstract data storage and computing service requests transparently taking into account the current load within the distributed system. The implementation relies on the standard middleware foundation CORBA and uses in the current version an interpreter based instantiation approach. For a more detailed description of the Amica infrastructure see [2].

3 The Amica Programming Model

A metacomputing application on top of Amica is composed of data storage and computation components which are directly related to the abstract data objects and metabricks of Amica. User-bricks allow additional application specific code to be integrated. A basic set of control flow and data flow connectors is used to specify and control component activation and interaction.

In the following we formally describe the organization of meta-applications as a hierarchical collection of interacting components with well defined properties and interfaces. This structural description is combined with a formal characterization of component behavior based on a process algebra. On this combination we build up a basic vocabulary for meta-application description.

3.1 Architecture Description

Despite the variety of existing software architecture description languages (ADLs) there is a considerable agreement about the role of structure in architecture description. Our description of metacomputing applications is based on ACME [5] which emerged from a joint effort of the architecture research community to provide a common intermediate representation for different ADLs.

Meta-applications are given as collections of components interconnected by connectors in a meta-application graph. The structural description is enriched by a behavioral characterization of components and connectors using the process algebraic description language Lotos [10].

Definition 1. A meta-application graph (MG) is given as a bipartite graph, which is characterized by a quintuple $G = (Nodes, I, E, Beh)$.

- The $Nodes$ of the graph consist of a set of components $Comp$ and a disjoint set of connectors $Conn$: $Nodes = Comp \cup Conn, Comp \cap Conn = \{\}$.
- The function $I : Nodes \rightarrow 2^{IP}$ associates each node with its set of interconnection points ($\in IP$). Interconnection points are subdivided into ports (for components) and roles (for connectors). ($IP = Ports \cup Roles$).
- The interconnection of components and connectors via their ports and roles is given by the relation $E \subseteq (Comp \times Ports) \times (Conn \times Roles)$.
- The function $Beh : Nodes \rightarrow LotosTerm$ associates each node to its behavioral description in form of a Lotos process algebra term.

For a node c in a meta-application graph with an interconnection point $p1$ and an associated set of actions $\{a_1, .., a_n\}$ (e.g. services needed or provided or events emitted at the interconnection point) the associated process term $Beh(c)$ defines a Lotos process c with parameter list $[p1\_a_1, .., p1\_a_n]$. This list is extended accordingly if multiple interconnection points are defined for a node. Interaction with other (node-) processes is exclusively possible via synchronization with these externally oberservable actions.

3.2 An Architectural Style for Amica Meta-applications

Each node and each interconnection point in a meta-application graph is an instance of a type from a set of predefined type definitions. These types are used to build up a basic vocabulary *Voc* to describe the architecture of a meta-application. This vocabulary along with a set of constraints is often called an architecture style [13].

Definition 2. A meta-application vocabulary Voc to build up (behavioral) meta-application graphs is given as a quintuple (NT, IPT, SC, BC) where NT is a set of node (component or connector) types, IPT is a set of interconnection point (port or role) types, SC is a set of structural constraints, and BC is a set of behavioral constraints.

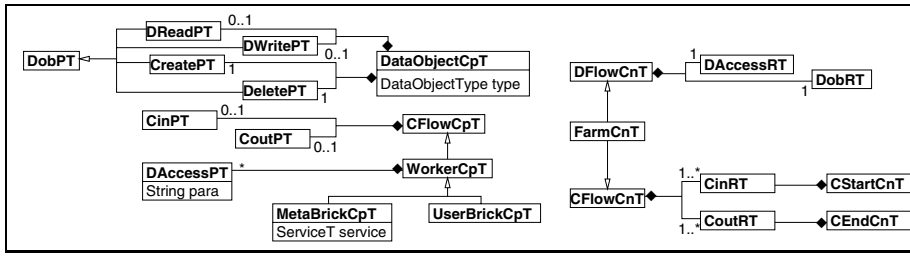


Fig.1. Basic meta-application vocabulary

The description of metacomputing applications on top of Amica is currently based on a simple architectural vocabulary which is illustrated in Fig. 1.

It contains component and connector types to characterize the flow of control and the flow of data, to describe farm parallelism, and to provide access to data storage and computing resources. For brevity we omit the structural and semantic constraints. In general, structural constraints are given by the associations in the class diagram and first order logic predicates. Semantic constraints currently are given in an action based temporal logic.

Data is stored in instances of the type `DataObjectCpT`. Its definition includes two port types for read-write access and two port types to create and delete object instances. The access to data object components is done via connectors of the type `DFlowCnT`. It provides roles for connection to data objects and to objects needing data access. Each component which is wired in control flow is an instance of type `CFlowCpT`. This type defines two interconnection points of type `CinPT,CoutPT`, characterizing incoming and outgoing control flow. Control flow components are connected to control flow connectors (type `CFlowCnT`). Different subtypes characterize connectors which split and combine control flow.

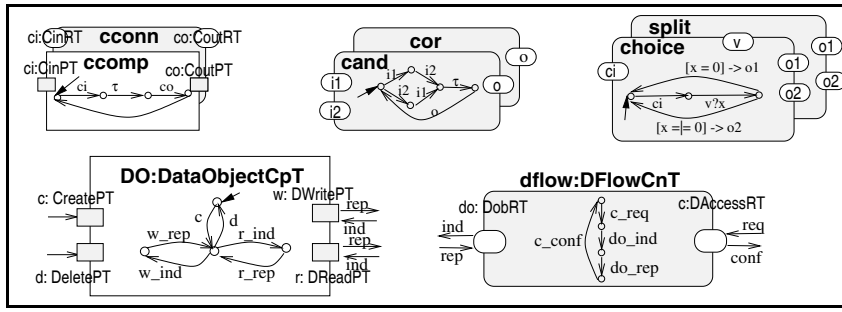
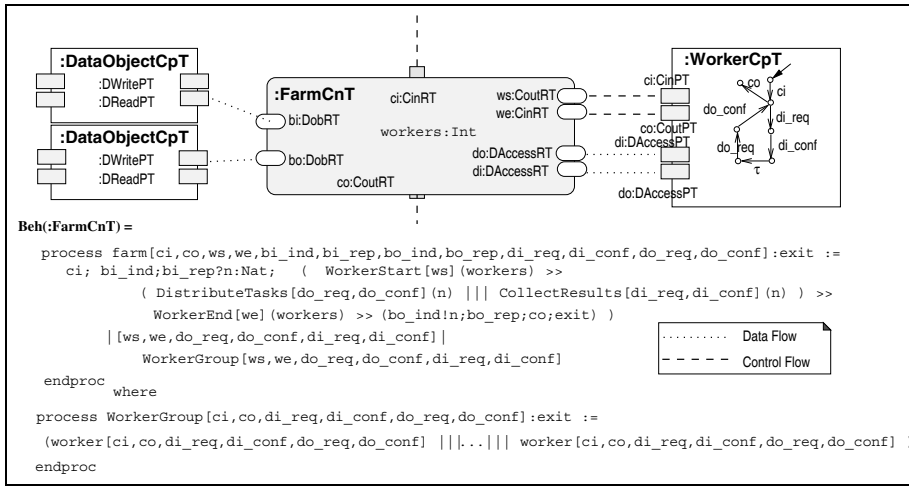


Fig.2. Basic data and control flow components and connectors

Examples of basic data and control flow components are given in Fig. 2. Their basic behavior is illustrated in the form of labeled transition systems. Thus control flow is simply propagated and data flow is based on a simple request–confirm protocol.

Instances of the special connector type `FarmCnT` are used to express and control "bag of task" like parallel computations. Farm connectors are used only in a well defined cooperation with data object and worker components, see Fig. 3. After started the farm reads in a bag containing the tasks to be distributed (using `bi:DobRT`) and starts a number of workers over `ws:CoutRT` (this number is given as a property value of the connector). Then the tasks are distributed to the workers using role `do:DAccessRT`. Thereafter the results are collected over `di:DAccessRT` and stored in a result bag data object attached to role `bo`. This description of the behavior corresponds directly to the Lotos description given as skeleton in Fig. 3.



Beh(:FarmCnT) =

```
process farm[ci,co,ws,we,bi_ind,bi_rep,bo_ind,bo_rep,di_req,di_conf,do_req,do_conf]:exit :=
   ci; bi_ind;bi_rep?n:Nat;   (  WorkerStart[ws](workers) >>
            ( DistributeTasks[do_req,do_conf](n) ||| CollectResults[di_req,di_conf](n) ) >>
             WorkerEnd[we](workers) >> (bo_ind!n;bo_rep;co;exit) )
         |[ws,we,do_req,do_conf,di_req,di_conf]|
            WorkerGroup[ws,we,do_req,do_conf,di_req,di_conf]
 endproc
        where
process WorkerGroup[ci,co,di_req,di_conf,do_req,do_conf]:exit :=
 (worker[ci,co,di_req,di_conf,do_req,do_conf] |||...||| worker[ci,co,di_req,di_conf,do_req,do_conf] )
endproc
```

Fig.3. Bag of task like parallelism based on farm connector and worker component

3.3 A Small Example

A simple exemplary composition based on our vocabulary defining a ray tracing meta-application is given in Fig. 4. Two data objects are involved, `scen` stores the three dimensional scenario and `pic` stores the generated picture.

In `init` these data objects are created and initialised. Then, the remote computation is started by a metabrick. In parallel the user can monitor the current state of the picture by a specialized user brick. Using the graphical

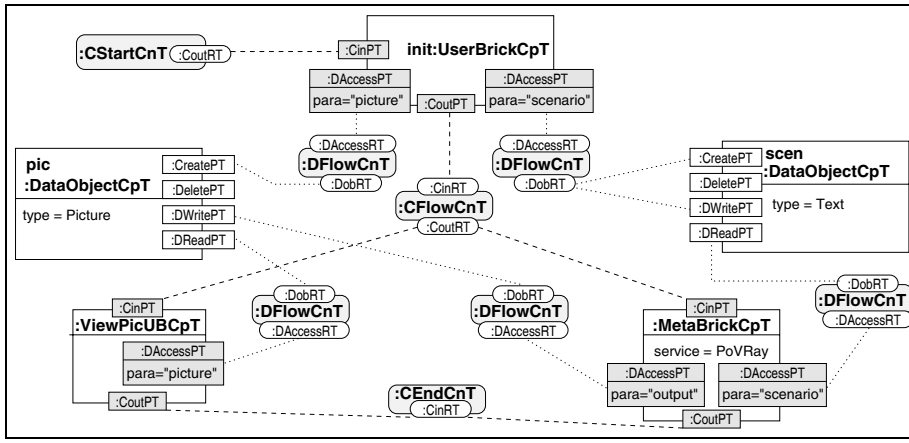


Fig.4. An exemplary application

front-end for ACME this application can be intuitively defined by drag-and-drop nodes out of our meta-application vocabulary. Another simple application to parallel simulation of mobile communication systems is described in [2].

4 Meta-application Execution and Formal Analysis

A meta-application graph is dynamically interpreted and mapped to the Amica metacomputer. Also automatically a global behavioral model of the meta-application is generated in a compositional way. This model can be analyzed and checked for design errors (e.g. resulting from component composition mismatches). These two steps, namely interpretation and behavioral model generation are discussed in the following section in more detail.

A simple meta-application interpreter is used to map our basic component and connector vocabulary to the services provided by Amica: Components of type `DataObjectCpT` and `MetaBrickCpT` are directly correlated to the data object and metabrick abstractions of basic data storage and computing services provided by Amica. The data store network and the broker mechanism of Amica relate these dynamically to the distributed data objects and bricks (created by brick factories). Data flow connectors and control flow connectors are interpreted to control component interaction and activation. In the special case of a farm connector (type `FarmCnT`) a specified number of workers is instantiated, the tasks contained in a bag are distributed and the results are collected.

The structure information given in the meta-application graph is used to compose the individual node behaviors to the combined overall system behavior. Composition is based on appropriate synchronization of component and connector actions. To correlate these actions we have to define a renaming operator

$\backslash\{(old_1\backslash new_1), .., (old_n\backslash new_n)\}$ which replaces all actions old_i_act over interconnection point names old_i to to new_i_act.

Given a behavioral meta-application graph $G = (Nodes, I, E, Beh)$ with $Nodes = Comp \cup Conn = \{CP_1, .., CP_N\} \cup \{CN_1, ..CN_M\}$ the associated overall system behavior is given by the parallel (full interleaving $|||$) composition of all component instances and also all connector instances. These two groups synchronize over all actions of their interconnection points ($||$). Thus the general structure of the overall system behavior description is given as the following process term scheme:

$$(NCP_1 \ ||| \ ... \ ||| \ NCP_N) \ \ || \ \ (NCN_1 \ ||| \ ... \ ||| \ NCN_M)$$

where

$$\begin{aligned} NCP_i = Beh(CP_i)\backslash\{(old\backslash new) \mid old \in I(CP_i) \wedge \ new = CN_j\_newp \\ \wedge \ \ attachedCN(CP_i, old) = (CN_j, newp)\} \\ NCN_i = Beh(CN_i)\backslash\{(old\backslash new) \mid old \in I(CN_i) \wedge new = CN_i\_old\} \end{aligned}$$

The function $attachedCN$ used above gives exactly one attached (connector, role) pair for a given component and port, that is attachments of multiple roles to a port are disallowed. In the case of multiple attachments of ports to a role, the above scheme implies an or semantics (the role-action synchronizes with an action associated with one of the attached ports). We have extended this to generally allow n-port to one role attachments with different semantics (e.g. and) given by the type of the role. This generation scheme was implemented in Java and we apply powerful formal analysis tools (the CADP tool set [4] and the model checker XTL [11]) for abstract meta-application property checks.

5 Related Work and Conclusion

Different approaches to build component based meta-applications are used in literature, ranging from simple data-flow models to general component frameworks. In WebFlow [9] (restricted) data flow graph models are proposed for component composition. In [14] a distributed component architecture toolkit is described for meta-application design on top of the Globus metacomputing infrastructure [3]. Also scripting languages can be used for meta-application description, see e.g. [12]. In contrast to these approaches having no or only a very restricted semantic foundation we use a more flexible and general formal process algebraic model combined with an architectural description of meta-applications using a well defined (and extendable) set of components and connectors.

Other approaches exist which are promoting the general idea of combining an architectural description with a formal behavioral model in the more general context of distributed software design. The ADL Darwin is combined with labeled transition systems in [7] to facilitate a compositional reachability analysis and in [1] CSP is used within the ADL Wright.

To conclude, we have described a formal model supporting component based meta-application development. A formal architecture description language was

combined with a process algebraic behavioral description to define a basic extendable vocabulary of component types for meta-application development. This vocabulary has been implemented on top of the Amica metacomputing infrastructure. Component compositions can automatically be checked based on well known state space analysis methods (e.g. model checking). Next steps include an extension of our vocabulary (including e.g. event propagation and handling) and an application to more complex problems. As we want to check not only functional but also performance properties of meta-applications we plan to use a stochastic process algebra description of component behavior. Also a behavioral model of the Amica infrastructure itself will be integrated in the future.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.

[2] T. Fink and S. Kindermann. First steps in metacomputing with Amica. In *Euromicro-PDP 2000*, pages 197–204. IEEE Computer Society, 2000.

[3] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[4] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien. CADP'97 - status, applications, and perspectives. In *Proceedings of 2nd COST 247 Int. Workshop on Applied Formal Methods in System Design*, 1997.

[5] D. Garlan, R.T. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON '97*, November 1997.

[6] D. Garlan and M. Shaw. *Software Architecture: Perspectives on an emerging Discipline*. Prentice Hall, April 1996.

[7] D. Giannakopoulou, J. Kramer, and S.C. Cheung. Behaviour analysis of distributed systems using Tracta. *Journal of Automated Software Engineering*, 6(1):7–35, January 1999. R. Cleaveland and D. Jackson, Eds.

[8] A. Grimshaw, A. Ferrari, G. Lindahl, and K. Holcomb. Metasystems. *Communications of the ACM*, 41(11), 1998.

[9] T. Haupt, E. Akarsu, and G. Fox. Webflow: a framework for web based metacomputing. In *HPCN Europe '99*, April 1999.

[10] ISO/IEC. Lotos — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, ISO — Information Processing Systems — OSI, Genève, September 1988.

[11] R. Mateescu and H. Garavel. Xtl: A meta-language and tool for temporal logic model-checking. In Tiziana Margaria, editor, *STTT'98 (Denmark)*, July 1998.

[12] R.P. Mc Cormack, J.E. Koontz, and J. Devaney. Seamless computing with WebSubmit. *Concurrency: Practice and Experience*, 11(15):946–963, 1999.

[13] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings COMPSAC*, 1997.

[14] J. Villacis, M. Govindaraju, D. Stern, A. Withaker, F. Berg, P. Deuskar, T. Benjamin, D. Gannon, and R. Bramley. Cat: A high performance, distributed component architecture toolkit for the grid. In *Proceedings of the High Performance Distributed Computing Conference*, 1999.

Javelin 2.0: Java-Based Parallel Computing on the Internet

Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{neary, evodius, joy, cappello}@cs.ucsb.edu

Abstract. This paper presents Javelin 2.0. It presents architectural enhancements that facilitate aggregating larger sets of host processors. It then presents: a branch-and-bound computational model, the supporting architecture, a scalable task scheduler using distributed work stealing, a distributed eager scheduler implementing fault tolerance, and the results of performance experiments. Javelin 2.0 frees application developers from concerns about complex interprocessor communication and fault tolerance among Internetworked hosts. When all or part of their application can be cast as a piecework or a branch-and-bound computation, Javelin 2.0 allows developers to focus on the underlying application.

1 Introduction

Our goal is to harness the Internet's vast, growing, computational capacity for ultra-large, coarse-grained parallel applications. By providing a portable, secure programming system, Java holds the promise of harnessing this large heterogeneous computer network as a single, homogeneous, multi-user multiprocessor [1]. Some research projects that are designed to exploit this include *Charlotte* [4], *Atlas* [3], *Popcorn* [6], *Javelin* [7], *Bayanihan* [12], *Manta* [13], *Ajents* [8], and *Globe* [2]. Javelin 2.0 is designed to achieve two goals: 1) Obtain the performance of a massively parallel implementation; 2) Provide a simple API, allowing designers to focus on a recursive decomposition/composition of the parallelizable part of the computation. The application programmer gets the performance benefits of massive parallelism, without adulterating the application logic with interprocessor communication details and fault tolerance schemes. The resulting code should run well on a set of processors that changes during execution. Javelin 2.0 handles all interprocessor communication and fault tolerance for the application programmer, when the parallelizable computation can be cast as a branch-and-bound (or piecework) computation. This is a broad class of computations. We focus here on 2 fundamental issues:

- *Scalable Performance* — If there is no niche where Java-based global computing outperforms existing multiprocessor systems, then there is no reason to use it. The architecture must scale to a higher degree than existing multiprocessor architectures, such as networks of workstations.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1231–1238, 2000.
© Springer-Verlag Berlin Heidelberg 2000

– *Fault tolerance* — An architecture that scales to thousands of hosts must be fault tolerant, particularly when hosts, in addition to failing, may dynamically disassociate from an ongoing computation.

Javelin 2.0 extends the piecework computational model to a branch-and-bound model, which is implemented using a weak form of shared memory that itself is implemented via the *pipelined RAM* [9] model of cache consistency. This shared memory model is strong enough to support branch-and-bound computation (in particular, bound propagation), but weak enough to be fast. Using this cache consistency model, we present a high-performance, scalable, fault tolerant Internet architecture for branch-and-bound computations, such as are used to solve NP-complete problems. For such an architecture to succeed, the architects must be diligently cognizant of the central technical constraint: On the Internet, *communication latency is large.*

2 Model of Computation

The branch-and-bound method, which generalizes the *piecework model of computation* [10], intelligently enumerates all feasible points of a combinatorial optimization problem: not all feasible solutions are examined. Branch-and-bound, in effect, proves that the best solution is found without necessarily examining all feasible solutions. The method successively partitions the solution space (*branches*), and prunes a subspace, when there is sufficient information to infer that none of the subspace's solutions are as good as a current solution (*bound*). (See Papadimitriou and Steiglitz [11] for a more complete discussion of branch-and-bound.) The computational model implies the following requirements: 1) Tasks (elements of the activeset) are generated during the host computation; 2) When a host discovers a new best cost, it propagates it to the other hosts; 3) Detecting termination in a distributed implementation requires knowing when all subspaces (children) have been either fully examined or killed. The challenge, in sum, is, with a *minimum of communication*, to enable: a) hosts to create tasks, which subsequently can be stolen; b) hosts to propagate new bounds rapidly to all hosts; c) the eager scheduler to detect tasks that have been completed or killed. The last item is needed not just for termination detection, but for fault tolerance, to determine which tasks need to be rescheduled.

3 Architecture

The Javelin 2.0 system architecture retains the basic structure of its predecessors, Javelin [7] and Javelin++ [10]. There are three system entities — clients, brokers, and hosts. A *client* is a process seeking computing resources; a *host* is a process offering computing resources; a *broker* is a process that coordinates the allocation of computing resources.

3.1 Javelin Broker Name Service

When a host (or client) wants to connect to Javelin, it first must find a broker that is willing to serve it. The JavelinBNS system is a scalable, fault tolerant directory service that enables the discovery of a nearby Javelin broker, without any prior knowledge of the broker network structure. It is designed not only to aid hosts who are searching for brokers, but also to aid brokers who are looking for neighboring brokers.

A JavelinBNS system consists of at least two fully replicated JavelinBNS servers. Each server is responsible for managing a list of available brokers, responding to broker lookup requests, and ensuring that the other JavelinBNS nodes contain the same information. The JavelinBNS system thus serves as an information backbone for the entire Javelin 2.0 system. Since the information stored for each broker is relatively small, the service will scale to a very large number of brokers. A small number of BNS servers will therefore be capable of administering thousands of broker entries, so a fully connected network of BNS servers will not be a bottleneck. At regular intervals, information is exchanged by the BNS servers. If a BNS server crashes and subsequently restarts, it can simply reload its tables with the information from its neighbors, thus providing for fault tolerance. Figure 1 shows the steps involved in a broker lookup operation.

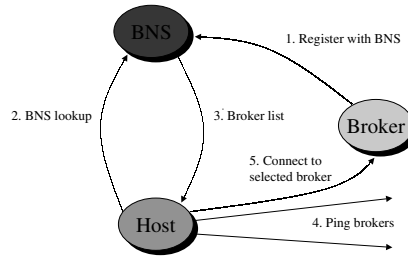


Fig. 1. JavelinBNS lookup sequence.

3.2 Broker Network & Host Tree Management

The topology of the broker network is an *unrestricted graph of bounded degree*. Thus, at any time a broker can only communicate with a constant number of other brokers. Similarly, a broker can only handle a constant number of hosts. If that limit is exceeded adequate steps must be taken to redirect hosts to other brokers. The bounds on both types of connection give the broker network the potential to scale to arbitrary numbers of participants. At the same time, the degree of connectivity is higher than in a tree-based topology.

When a host connects to a broker, the broker enters the host in a logical tree structure. The top-level host in the tree will not receive a parent; instead it will

later become a child of the client. This way, the broker maintains a *preorganized tree of hosts* which are set on standby until a client becomes active. When a client connects, or client information is remotely received from a neighboring broker, the whole tree is activated in a single operation and the client information is passed to the hosts. Brokers can individually set the branching factors of their trees, and decide how many hosts they can administer. In case of a host failure, the failed node is detected by its children and the broker restructures the tree in a heap-like operation (for details, see [10]).

4 Scalable Computation & Fault Tolerance

4.1 The Scheduler

The fundamental concept underlying our approach to task scheduling is *work stealing*, a distributed scheduling scheme made popular by the Cilk project [5]. Work stealing is entirely demand driven — when a host runs out of work it requests work from some host that it knows. Work stealing balances the computational load, as long as the number of tasks is high relative to the number of hosts — a property well suited for adaptively parallel systems.

In Javelin 2.0, tasks get split in a double-ended task queue until a certain minimum granularity — determined by the application — is reached. Then, they are processed. When a host runs out of local tasks, it selects a neighboring host and requests work from that host. Since the hosts are organized as a tree, the selection of the host to steal work from follows a deterministic algorithm based on the tree structure. Initially, each host retrieves work from its parent, and computes one task at a time. When a host finishes all the work in its deque, it attempts to steal work, first from its children, if any, and, if that fails, from its parent. This strategy ensures that all the work assigned to the subtree rooted at a host gets done before that host requests new work from its parent. Work stealing helps each host get a quantity of work that is commensurate with its capabilities. The client is the root of its tree of hosts.

4.2 Shared Memory

For branch-and-bound computation, only a *small amount* of shared memory is needed and a *weak shared memory model* suffices. The small amount is because only one integer is needed to represent a solution's *cost*. The weak model suffices because if a host's copy of best cost is stale, correctness is unaffected. Only performance may suffer — we might search a subspace that could be pruned. It thus suffices to implement the shared memory using a pipelined RAM (aka PRAM) model of cache consistency. This weak cache consistency model can be implemented with scalable performance, even in an Internet setting.

There are several methods to propagate bounds among hosts. We use the following: When a host discovers a solution with a better cost than its cached best cost, it sends this solution to the client. If the client agrees that this indeed

is a new best cost solution (it may not be, due to certain race conditions), it updates its cached best cost solution, and "broadcasts" the new best cost to its entire tree of hosts. That is, it propagates the new best cost to its children, who in turn propagate it to their children, level by level down the host tree.

4.3 Fault Tolerance

Eager scheduling *re*schedules a task to an idle processor in case its result has not been reported. It was introduced and made popular by the Charlotte project [4], and also has been used successfully in Bayanihan [12]. Javelin++ [10] also uses eager scheduling to achieve fault tolerance and load balancing. It efficiently and relentlessly progresses towards the overall solution in the presence of host and link failures, and varying host processing speeds.

The Javelin 2.0 eager scheduler is located on the client. Although this may seem like a bottleneck with respect to scalability, it is not, as we shall explain below. Eager scheduling, however, is more challenging for branch-and-bound computation (as compared to piecework computation). Besides detecting *positive results*, (i.e., new best cost solutions), the eager scheduler must detect *negative results*: solution [subspaces] that have been examined and do not contain a new best cost solution, and solution subspaces that have been pruned. Performance, though, requires avoiding unnecessary communication and computation.

In a branch-and-bound computation, the size of the feasible solution space is *exponential* in the size of the input. In principle, the algorithm may need to examine all of these exponentially many feasible solutions to find the minimum cost solution. In practice, a partial solution, p, is "killed" (a subspace is pruned) when the lower bound on the cost of any feasible solution that is an extension of p *must be* more costly than the currently known minimum cost solution. The algorithm nonetheless must gather sufficient information to detect that the minimum cost solution has indeed been found. This implies that killed nodes and sub-optimal solutions must be detected by the eager scheduler. If a separate communication is required to detect each such event, the overall quantity of communication would nullify the benefits of parallelism. We cope with this communication overload by aggregating portions of the search space into atomic tasks, and similarly aggregating negative results into one communication per atomic task. This lets the eager scheduler know that this part of the problem tree has been searched, and hence need not be rescheduled. The number of negative communications consequently is equal to or less than the number of atomic tasks. In practice, it is much less than the number of atomic tasks; many are killed. We can adjust the computation/communication ratio by adjusting the size of atomic tasks, in order to decrease the overall run time. Performance is quite sensitive to atomic task size, so finding good size values is important. For performance reasons, we balance the computational *size* of the hosts' atomic tasks with the client's computation of result handling and eager scheduling, so that neither the client nor the hosts have to wait for one another. Additionally, we want the number of atomic tasks to be much larger than the number of hosts, to keep them all well utilized, even when some are *much* faster than others.

5 Experimental Results

All experiments were run in campus computer labs under a typical workload. The heterogeneous test environment consists of 4 Sun Enterprise 450 dual/quad-processors with a processor speed of 400 MHz; 49 Celeron 466/500 MHz processors; and a Beowulf cluster of 42 nodes, with 6 Pentium III 500 MHz quad-processors, and 36 Pentium II 400 MHz dual processors. The cluster is running Red Hat Linux 6.0. All other machines are running Solaris 2.7. We used JDK 1.2 with active JIT for our experiments.

We tested the performance of Javelin 2.0 with a TSP application. The test graphs are complete, undirected, weighted graphs of 22 and 24 nodes, with randomly generated integer edge weights. These graphs are complex enough to justify parallel computing, but small enough to enable us to run tests in a reasonable amount of time. The 22-node graph took approximately 3 hours to process on a Sun E450. The 24-node graph took just under 10 hours on the same processor.

The term "speedup" is somewhat confusing here. Traditionally, speedup is measured on a dedicated multiprocessor, where all processors are homogeneous in hardware and software configuration, and varying workloads between processors do not exist. Thus, speedup is well defined as T_1/T_p, where T_1 is the time a program takes on one processor and T_p is the time the same program takes on p processors. Therefore, strictly speaking, in a heterogeneous environment like ours the term speedup cannot be used anymore. Even if one tries to run tests in as homogeneous a hardware setup as possible, the varying workloads on both the OS and the network can amount to big differences in the individual performance of hosts. However, from a practical standpoint, a user running an application on Javelin 2.0 with a large set of hosts will definitely see "speedup"; the application will run faster than on a single machine. We will use the term *practical speedup* to distinguish between the two scenarios. In the following, we may omit the word "practical" when the meaning is clear from the context.

We now give a more formal definition of our notion of practical speedup: Let $M_1, \ldots, M_k$ denote k different processor types. Let $T_1(i)$ denote the time to complete the problem using 1 processor of type M_i. Conventional speedup, using p processors of type M_i can be defined as $T_1(i)/T_p(i)$. To compute speedup when we have more than one type of processor, we generalize this formula. Let a problem be solved concurrently using k types of processors, where there are p_i processors of type M_i: The total number of processors is $p = p_1 + \cdots + p_k$. Let $T_p(p_1, \ldots, p_k)$ denote the execution time when using this mix of p processors. We define a *composite base* case that reflects this mix of processors:

$$T_1(p_1, \ldots, p_k) = \frac{p_1 T_1(1) + \cdots + p_k T_1(k)}{p_1 + \cdots + p_k}.$$

Finally, we define the speedup S as

$$S = T_1(p_1, \ldots, p_k)/T_p(p_1, \ldots, p_k).$$

While this definition does not incorporate machine and network load factors, it does reflect the heterogeneous nature of the set of machines.

Figure 2 shows the speedup we measured in our experiments and calculated according to the above formula. For the 22-node graph, speedup was superlinear at first, until it topped out at 77.26 for 100 hosts, when communication became a significant bottleneck. To observe superlinear speedup for the parallel TSP is quite common, due to the inherent irregularity of the input graph. The results for the 24-node graph illustrate this even further. Here, speedup was nowhere near as good, reaching only 23.35 for 80 hosts. However, the curve still shows a steady rate of improvement, and the larger graph has the potential to scale better due to its higher computation complexity.

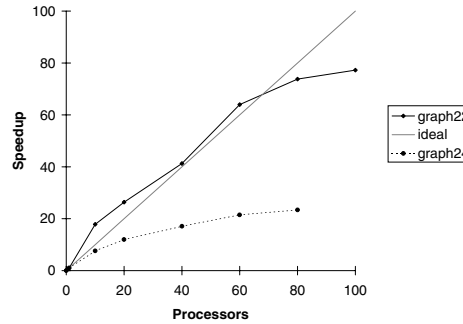


Fig. 2. Practical Speedup for TSP on Javelin 2.0.

To sum up, a graph that took about 3 hours to calculate on a single computer took just under 3 minutes on 100 processors under their normal workloads. These results are encouraging, although they need to be evaluated with different input graphs and more hosts.

6 Conclusion

To enlarge the set of applications that can benefit from Javelin, Javelin 2.0 extends Javelin++'s piecework model of computation to a branch-and-bound model. The technical challenge is to implement a distributed shared memory that enables hosts to share bounds. We implemented the pipelined RAM model of cache consistency among hosts sharing the bound. Our experiments indicate that *limited* use of this weak shared memory poses *no* performance problem.

To facilitate aggregating large numbers of hosts, Javelin 2.0 enhances host registration: The host can request the broker name system to return k broker names, where k is chosen by the host. Currently, the host then pings these brokers to discover the "nearest". In Javelin 2.0, with but one Java RMI call on a broker, a client gets a handle to the broker's entire *preorganized* host tree. Other brokers convey their host trees with a similar economy of communication.

The TSP experiments suggest that branch-and-bound can be sped up efficiently, even with large numbers of Internetworked hosts. Many combinatorial optimization versions of NP-hard problems are solved with branch-and-bound.

Our distributed deterministic work stealing scheduler integrates smoothly, not only with bound caching, but also with the distributed eager scheduler, which provides essential fault tolerance.

References

[1] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] A. Bakker, M. van Steen, and A. S. Tanenbaum. From Remote Object to Physically Distributed Objects. In *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Cape Town, South Africa, Dec. 1999.

[3] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*, pages 207–216, Santa Barbara, CA, July 1995.

[6] N. Camiel, S. London, N. Nisan, and O. Regev. The POPCORN Project: Distributed Computation over the Internet in Java. In *6th International World Wide Web Conference*, Apr. 1997.

[7] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, Nov. 1997.

[8] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, June 1999.

[9] Lipton and Sandberg. PRAM: A scalable shared memory. Technical report, Princeton University: Computer Science Department, CS-TR-180-88, Sept. 1988.

[10] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience, to appear*, 2000.

[11] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[12] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5-6):675–686, Oct. 1999.

[13] R. van Nieupoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, June 1999.

Data Distribution for Parallel CORBA Objects

Tsunehiko Kamachi[1], Thierry Priol[2], and Christophe René[2]

[1] C&C Media Research Laboratories, NEC Corporation, 4-1-1 Miyazaki, Miyamae-ku, Kawasaki, Kanagawa 216-8555 Japan
[2] IRISA/INRIA, Campus de Beaulieu - 35042 Rennes Cedex, France

Abstract. The design of application for Computational Grids relies partly on communication paradigms. In most of the Grid experiments, message-passing has been the main paradigm either to let several processes from a single parallel application to exchange data or to allow several applications to communicate between each others. In this article, we advocate the use of a modern approach for programming a Grid. It is based on the use of distributed objects, namely parallel CORBA objects. We focus our attention on the handling of distributed data within parallel CORBA objects. We show some performance results that were obtained using a NEC Cenju-4 parallel machine connected to a PC cluster.

1 Introduction

With the availability of high-performance networking technologies, it is nowadays feasible to couple several computing resources together to offer a new kind of computing infrastructure that is called a *Computational Grid* [4]. Such system can be made of a set of heterogeneous computing resources that are interconnected together through multi-gigabit networks. Software infrastructures, such as Globus [3] or Legion [6], provide a set of basic services to support the execution of distributed and parallel programs. One of the problem that arises immediately is how to program such a computational Grid and what is the most suitable communication model for Grid-enabled applications ? It is very tempting to extend existing message-passing libraries so that they can be used for distributed programming. We believe that this approach cannot be seen as a viable solution for the future of Grid Computing. Instead, we advocate an approach that allows the combination of communication paradigms for parallel and distributed programming. This approach, called PaCO, is based on an extension to a well known and mature distributed object technology, namely CORBA.

2 Communication within a Computational Grid

There exist two main approaches to communicate within a computational grid. The first approach is to allow the execution of a parallel code over heterogeneous machines taking benefit of the available computing resources. Research works have recently lead to extend existing message passing libraries to be able

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1239–1249, 2000.
© Springer-Verlag Berlin Heidelberg 2000

to exchange data between heterogeneous computing resources, such as MPICH-G [5], PACX [1] or PLUS [11]. A parallel code, based on one of these communication libraries, can be executed on a Grid with some minor modifications. We think that such approach is relevant since the purpose of these Grid-enabled communication libraries is to allow parallel programming at a larger scale. Such libraries can also be used to connect several parallel codes together to perform coupled simulations. It constitutes the second approach. The objective is to solve new kind of problems that were not affordable due to the lack of computing resources. The aggregating of computing resources may allow the simulation, in a shorter time frame, of complex manufactured products for which different physical behaviors have to be taken into account (structural mechanics, computational fluid dynamics, electromagnetism, noise analysis, etc...). Moreover, distributed execution of simulation codes are nowadays imposed by the way the industrial companies work together to design manufactured products. It requires that each company, participating to the design of a manufactured product, to contribute to the simulation of the whole product by providing access to its own simulation tools. However, a company is often reluctant to give both its simulations tools and the necessary simulation data to other companies (that may act as competitors later on). Therefore, there is a strong need to have part of the simulation of the whole product performed on their own computing resources to avoid the exchange of confidential data (i.e. the model of the object to be simulated). Thus, there is a clear need to have a mechanism to let simulation codes to communicate together. However, such mechanism requires that it is capable of both transferring data and control efficiently between codes.

We think that message-passing is not suitable to connect several parallel codes together. Indeed, message-passing was mainly designed for parallel programming and not for distributed programming; it is mainly to transfer data but not the control. As for instance, if one code would like to call a particular function into another code, this latter has to be modified in such a way that a message type is associated to this particular function. Such modification requires a deep understanding of the code. Moreover, entry points in a code are not really exposed to potential users that would like to include such code into its application. Communication paradigms, such as RPC or distributed objects, offer a much more attractive solution since the transfer of control is implemented by remote invocation that is as simple as calling a function or a method. However, they are not suitable for parallel programming due to their higher communication cost. It is thus clearly difficult to have a single communication paradigm for the programming of computational grids. We advocate an approach, like others [8], that consists in merging several communication paradigms in a coherent way so that they fit the requirements mentioned previously.

The remainder of this paper is structured as follows. Section 2 discusses communication issues for Computational Grids. Section 3 gives an overview of the parallel CORBA object concept. Section 4 describes data redistribution within a parallel CORBA object. Section 5 provides some experimental results. Finally, we conclude in section 6 by laying the grounds for future enhancement.

3 Overview of Parallel CORBA Object

CORBA is a specification from the OMG (Object Management Group) to support distributed object-oriented applications. CORBA acts as a *middleware* that provides a set of services allowing the distribution of objects among a set of computing resources connected to a common network. Transparent remote method invocations are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. An object interface is specified using the Interface Definition Language (IDL). An IDL file contains a list of operations for a given object that can be invoked remotely. An IDL compiler is in charge of generating a *stub* for the client side and a *skeleton* at the server side. A stub is simply a proxy object that behaves as the object implementation at the server side. Its role is to deliver requests to the server. Similarly, the skeleton is an object that accepts requests from the ORB and delivers them to the object implementation. The concept of parallel CORBA object[1]

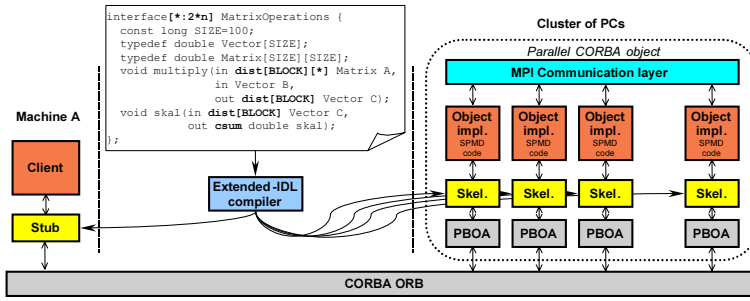


Fig. 1. Encapsulation of MPI-based parallel codes into CORBA objects.

is simply a collection of identical CORBA objects as shown in figure 1. It aims at encapsulating a MPI code into CORBA objects so that a MPI code can be fully integrated into a CORBA-based application. Our goal is to hide as much as possible of the problems that appear when dealing with coarse-grain parallelism on a distributed memory parallel architecture like a cluster of PCs. However, this is done without entailing a lost of performance when communicating with the MPI code. First of all, the calling of an operation by a client will result in the execution of the associated method by all objects belonging to the collection at the server side. Execution of parallel objects is based on the SPMD execution model. This parallel activation is done transparently by our system. Data distribution between the objects belonging to a collection is entirely handled by the system. However, to let the system to carry out parallel execution and data distribution between the objects of the collection, some specifications have to be added to the component interface. A parallel object interface is thus described

[1] we will use parallel object from now on

by an extended version of IDL, called Extended–IDL as shown in figure 1. It is a set of new keywords (in bold in the figure), added to the IDL syntax[2], to specify the number of objects in the collection, the shape of the virtual node array where objects of the collection will be mapped on, the data distribution modes associated with parameters and the collective operations applied to parameters of scalar types.

4 Data Redistribution in a Parallel Object

Application programmers have to specify, for each operation, parameters that have to be distributed among the collection and how they are distributed. Since they can define data distribution for a parameter in each parallel object differently, parameter values need to be redistributed. This problem is made difficult due to various configurations at both the client and the server side (data distribution modes, distributed dimension, object collection size or its virtual shape). It is thus necessary to provide a data redistribution mechanism, as part of the operation invocation, to facilitate the coupling of several parallel objects. Furthermore, while users are responsible for data distribution management within a parallel object, data redistribution should have to be handled by a runtime system in order to hide all the operations associated with the invocation of operations from or to a parallel object.

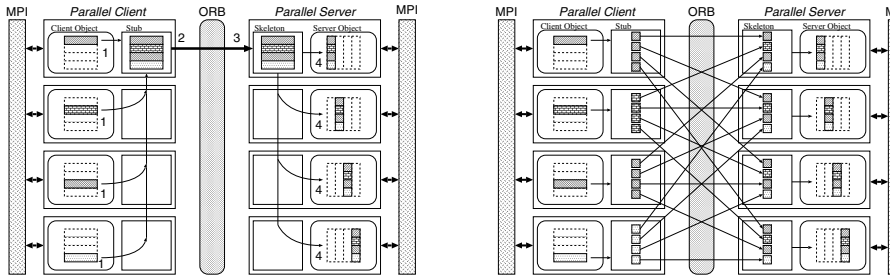


Fig. 2. Master/slave approach (a) Through the CORBA ORB (b)

4.1 Design Considerations

The most obvious way to perform data redistribution is to use gather and scatter operations at the client and the server sides. Figure 2-a illustrates such a technique for an operation invocation with a `in` parameter array. Four steps are required: first, one of the stubs gathers distributed data from the client objects using the MPI communication layer (1). Then, it invokes one of the server objects

[2] A more complete description of these extensions is given in [10, 12]

and sends gathered data to it through the CORBA ORB (2). During the third step, the skeleton of the activated server object receives the data from the ORB (3). Finally, the skeleton activates the remaining server objects and scatters data to them using MPI (4). Although this technique is simple to implement, it has some severe drawbacks. The gathering and scattering of data values associated with distributed parameters do not offer a good scalability when the number of objects increases. Data transfer between two collections is serialized through only one object. Furthermore, the gathering of data by one stub is memory consuming.

To avoid this problem, we need to incorporate both a parallel invocation of operations and a data redistribution strategy into parallel objects. One possible approach is to let client objects to send data values of the distributed parameters to the server objects through the ORB as shown in figure 2-b. In this approach, each client splits its own data according to the data distribution at the server side, and sends them to the relevant server objects directly. On the server side, each object receives pieces of data which it should own from several client objects. This approach suffers from a higher number of ORB requests compared to the master/slave approach. Since the ORB is usually much more slower than message-passing layers, we have to keep the number of requests as low as possible. Taken into account these remarks, we propose the following technique. Data

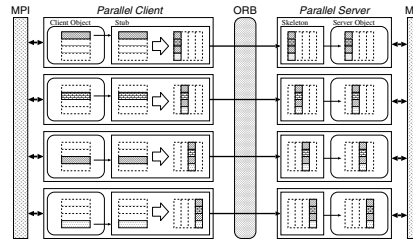


Fig. 3. Data redistribution in the client stub.

redistribution is performed by the client side or the server side before or after the sending of the request associated with the operation invocation. Figure 3 shows the case of data redistribution at the client side. Accordingly, all communications needed for the redistribution are carried out over the high-speed network of the parallel machines on which the parallel object, acting as the client, is running. The reason why we incorporated data redistribution into both the client and server sides is to obtain the maximum performance from the available communication resources. This allows us to select the most suitable place to perform data redistribution depending on the performance of the network at the client or the server side. More precisely, when data redistribution is performed at the client side, it is handled by the stubs that are aware of the data distribution mode of both the client and the server. First, all the stubs exchange their own data using the MPI communication layer to prepare data meeting the distribu-

tion mode at the server object, then each stub sends the redistributed data to the relevant server object through the ORB. When the data redistribution is performed by the server, the client has to provide to the server extra information associated with the distribution of the parameter along with the parameter data values. Such information includes, for each distributed parameter, the distribution mode and the distributed dimension; the virtual shape of the objects is also part of this information. When the skeleton receives such information, it redistributes the parameter data values to adapt the client data mapping to the server one.

4.2 Implementation

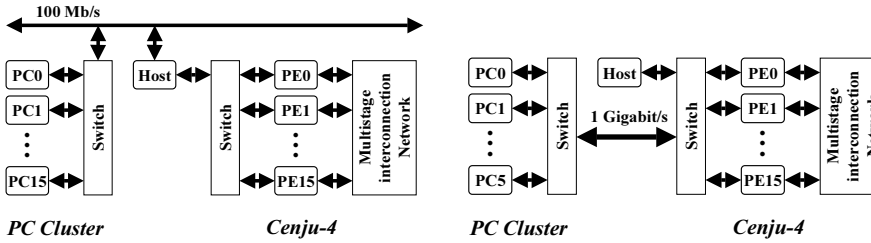
The generation of the stub code is rather complex due to various possibilities at the client side. As for instance, a client can be either a standard CORBA object or a parallel object. In the later case, data that have to be sent when invoking an operation on a parallel object are distributed among the objects of the client collection. Therefore, one possibility is to perform the data redistribution by the stub, using a data redistribution library, so that it fits the one at the server side. Another modification to the stub code concerns the invocation mechanism. Since the number of objects of the collection at the client side does not often coincides with the one at the server side, we added a mechanism that associates an object of the client collection to one of the object of the server collection. When there is only one object at the client side, the stub generates a request for each object of the server collection. Similarly, when there is only one object in the server collection, one object of the client collection is associated with this single object. As data redistribution has been done by the stub, the skeleton performs roughly the same work as a standard skeleton. It is worth mentioning that in this situation, skeletons do not need to communicate between each other within the server collection.

Another possibility is to perform data redistribution by the skeleton instead of the stub. In that case, the modification of the stub code generation is very simple. Each object of the client collection sends the data it owns to another object of the server collection. Before sending this data, the stub includes, for each parameter, data distribution information at the client side. It can then call a data redistribution library providing both the client and the server data distribution mode. Once redistribution is performed, the skeleton invokes the implementation method as a standard skeleton. If the parameter has an `inout` or `out` attribute, the skeleton builds the reply where it puts distributed data values following data distribution information sent by the client. It uses again the data redistribution library. This approach has the drawback of adding extra information (data distribution information) to the data sent by the client to the server. Moreover, such technique cannot be used when a sequential object has to invoke a method implemented by a parallel object. In such case, the client has to set up a request for each object of the server collection and thus has to distribute the data to each object of the collection.

To avoid implementing a new data redistribution library, we decided to adapt our stub and skeleton code generation process in such a way that we can exploit existing libraries. These libraries were developed for the High Performance Fortran (HPF) compilation system such as the NEC HPF/SC[7] and the GMD Adaptor system[2]. These two systems support all patterns of data redistribution in the scope of the HPF-1.1 specification. Since our extension for describing data distribution can be seen as a subset of HPF-1.1, all data redistribution patterns are covered. However, there are some limitations in our current implementation due to the difference of execution model between HPF and a parallel object. These data redistribution libraries are intended to be used within stand alone parallel program in which the number of processes is constant. However, we want to use these libraries to reorganize data when two parallel objects communicate. Such parallel programs may run on different number of processors and therefore we may have to redistribute data between two parallel objects that run on different number of processors. Moreover, such libraries were intended to be used with *Fortran* programs whereas we are using *C++*. Therefore, using these libraries requires extra memory copy operations to map *C++* arrays to *Fortran arrays*.

5 Experimental Results

We performed several experiments using two platforms (Figure 4); namely, the NEC distributed-memory parallel computer Cenju-4[9] and a PC cluster. The Cenju-4 has 16 PEs (processing elements) connected via a multistage interconnection network as well as a 100 Mb/s Ethernet network. Each PE consists of



(a) Using a 100 Mb/s Ethernet network (b) Using a 1Gb/s Ethernet network

Fig. 4. Experimental environment.

a 200MHz VR10000 RISC microprocessor. The PC cluster is a set of PCs connected to a 100 Mb/s Ethernet network. Each PC is equipped with two 450MHz Pentium III processors running Linux. ORB communications between the PEs of the Cenju-4 and the PCs of the PC cluster can go either through the Cenju-4 host machine using a 100 Mb/s network or through a 1Gb/s network. For the

experiment, we used the DALIB redistribution library [2]. The MPI communication layer was implemented on the multistage interconnection network on the Cenju-4 and on the fast Ethernet network on the PC cluster. In order to illustrate the effectiveness of our approach, we experimented a simple code; a parallel client issues a single invocation operation to a parallel server with a two dimensional distributed array parameter (long) with a `in` attribute. Distribution of the matrix at the client side follows a `[BLOCK][*]` distribution mode whereas at the server side the matrix to be distributed using a `[*][BLOCK]` distribution mode.

5.1 Comparison with the Master/Slave Approach

In this experiment, we ran both a parallel client and a parallel server on the PC cluster to compare the performance of the master/slave approach with that of our approach. Results presented in Figure 5 make it evident that our approach provides a scalable solution as compared with the master/slave approach. The other point we can observe is that a distinct difference between the performance of the client side redistribution and that of the server side redistribution cannot be seen. This result tells us the overhead of sending extra data in case of the server side redistribution is insignificant with our experimental environment.

5.2 Redistribution at the Client versus the Server

We measured the performance of a single operation invocation similar to what have be done to compare the master/slave approach with the parallel object one. However, this time we mapped the client onto the PC cluster and the server onto the Cenju-4. Data redistribution is performed either by the stub (using the MPI layer with the Ethernet network of the PC Cluster) or by the skeleton (using the MPI layer with the multistage network of the Cenju-4 parallel system). Results are presented in Figures 6. They were obtained by using either the 100 Mbit/s or the 1 Gbit/s Ethernet network. The time associated with the ORB communication, the memory copy and the data redistribution in the invocation are measured separately. ORB communication time corresponds to

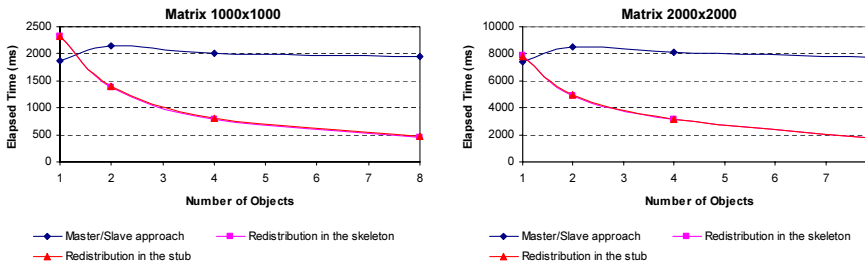


Fig. 5. Comparison our approach with the master/slave approach.

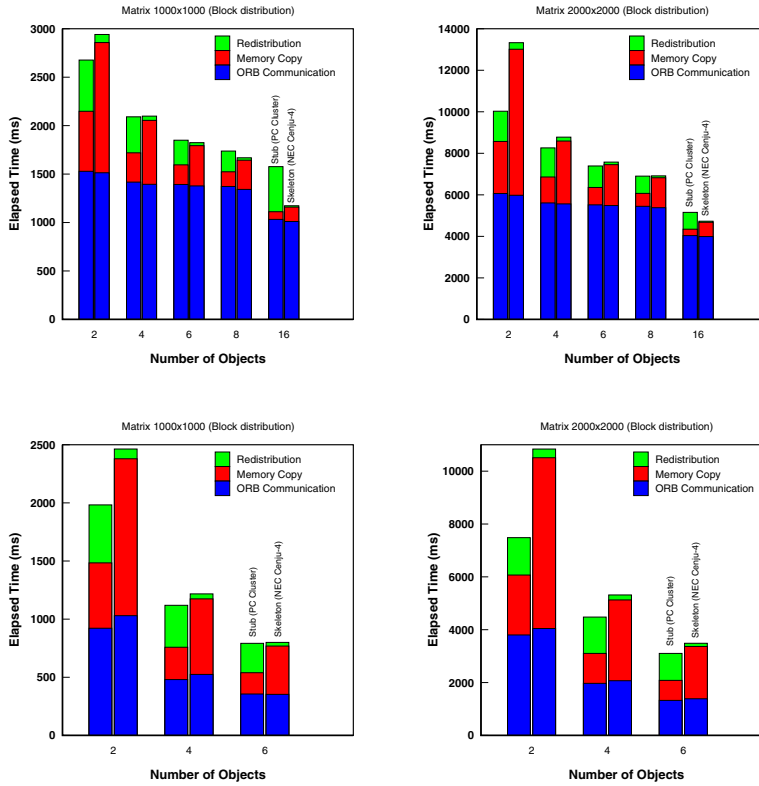


Fig. 6. Comparison of communication costs (top: 100 Mbit/s, bottom: 1 Gbit/s).

the invocation time without redistribution and memory copy. We cannot see a significant difference in the two test cases (data redistribution within the stub or the skeleton). However, as compared with the results measured within the PC cluster, the ORB communication time between the Cenju-4 and the PC cluster is slow. In addition, when using the 100 Mbit/s Ethernet network, it does not provide a good speedup when number of objects increases. This is because, as shown in Figure 4, all the ORB communications between the PEs in the Cenju-4 and the PCs in the PC cluster have to go through the Cenju-4 host computer. If the PEs are connected to the network directly (using the 1 Gbit/s Ethernet network), the performance and the speedup ratio is improved.

To handle both distributed data and the information about its distribution mode, we introduced a new data structure, called `darray` [12], based on the CORBA `sequence` data structure. Unlike an array, data in the `darray` is stored non-continuously into the memory if the `darray` realizes an array which has more than two dimensions in the same way as the `sequence`. Therefore, two memory

copy operations are required in the redistribution process, that is, copying data from `darray` structure to a *Fortran* array before redistribution and copying redistributed data from the *Fortran* array to `darray` structure after redistribution. Moreover, since the difference of memory mapping scheme between *C++* and *Fortran* arrays forces this memory copy by element, its overhead increases. The results clearly show that this memory copy causes serious overhead. In addition, we see from the figure that the parallel machines provide quite different results due mainly to the performance of the processors and the memory hierarchy that equippe each computing node. Consequently, as the Cenju-4 suffered from overhead of the memory copy, the PC cluster achieves better performance of the invocation in total, even if the Cenju-4 provides good performance for the data redistribution.

Redistribution time is the communication time for exchanging data to perform the redistribution using the MPI interface. Experiments shows that the time of redistribution on the Cenju-4 is up to 9 times faster than that on the PC cluster. However, as compared with the overhead of the memory copy, this performance difference gives less impact to the total time of the invocation.

6 Conclusion and Future Works

This paper discusses the implementation of data redistribution within a parallel CORBA object. We implemented a capability of performing data redistribution within parallel object in both a stub and a skeleton. This allows programmers to obtain the maximum performance under their distributed computing environment. In our current implementation, the selection is performed at compile time by specifying Extended-IDL compiler options. This means that programmers are responsible for deciding which side should perform redistribution. Although it is important to provide means to control data redistribution to the programmers, it is usually difficult for them to know which side provides better performance. This is because they have to take into account a lot of factors related to characteristics of data redistribution and their underlying computing environment (communication network, memory hierarchy and processor). In addition, the fact that such factors can vary at run-time due to the network contention makes this problem much harder. In order to relieve programmers from the pains of making such decision, as well as to provide the maximum performance automatically, we are developing a run-time service system to manage static and dynamic system information during the execution of parallel objects. This information will be used do decide at runtime the best place to carry out data redistribution.

Acknowledgments We would like to thank Satoshi Goto and Toshiyuki Nakata for their continuous valuable advice. This work was carrying out within the INRIA-NEC collaboration framework under contract 099C1850031308065.

References

[1] T. Beisel, E. Gabriel, and M. Resch. An extension to MPI for distributed computing on MPPs. *Lecture Notes in Computer Science*, 1332, 1997.

[2] T. Brandes and F. Zimmermann. Adaptor — A transformation tool for HPF programs. In *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3*, pages 91–96, April 1994.

[3] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[4] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infracstructure*. Morgan Kaufmann Publishers, Inc, 1998.

[5] Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, November 1998.

[6] A. S. Grimshaw, W. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 1(40):39–45, January 1997.

[7] T. Kamachi, K. Kusano, K. Suehiro, Y. Seo, M. Tamura, and S. Sakon. Generating realignment-based communication for HPF programs. In *10th International Parallel Processing Symposium*, 1996.

[8] K. Keahey and D. Gannon. Developing and Evaluating Abstractions for Distributed Supercomputing. *Cluster Computing*, 1(1):69–79, May 1998.

[9] T. Nakata, Y. Kanoh, K. Tatsukawa, S. Yanagida, N. Nishi, and H. Takayama. Architecture and software environment of parallel computer cenju-4. *NEC Research & Development*, 39(4):385–390, October 1998.

[10] T. Priol and C. René. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing. In *Euro-Par'98*, pages 1114–1122, September 1998.

[11] A. Reinefeld, J. Gehring, and M. Brune. Communicating across parallel message-passing environments. *Journal of Systems Architecture*, 44:261–272, 1998.

[12] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, August 1999.

Topic 20
Parallel I/O and Storage Technology

Rajeev Thakur, Rolf Hempel, Elizabeth Shriver, and Peter Brezany

Topic Chairpersons

Introduction

In recent years, it has become increasingly clear that the overall time to completion of parallel applications may depend to a large extent on the time taken to perform I/O in the program. This is because many parallel applications need to access large amounts of data, and although great advances have been made in the CPU and communication performance of parallel machines, similar advances have not been made in their I/O performance. The densities and capacities of disks have increased significantly, but improvement in performance of individual disks has not followed the same pace. For parallel computers to be truly usable for solving real, large-scale problems, the I/O performance must be scalable and balanced with respect to the CPU and communication performance of the system.

The parallel I/O and storage research community is pursuing research in several different areas in order to solve the problem. Active areas of research include disk arrays, network-attached storage, parallel and distributed file systems, theory and algorithms, compiler and language support for I/O, runtime libraries, reliability and fault tolerance, large-scale scientific data management, database and multimedia I/O, realtime I/O, and tertiary storage. The MPI-IO interface, defined by the MPI Forum as part of the MPI-2 standard, aims to provide a standard, portable API that enables implementations to deliver high I/O performance to parallel applications.

The Parallel I/O Archive at Dartmouth, `http://www.cs.dartmouth.edu/pario` is an excellent resource for further information on the subject. It has a comprehensive bibliography and links to various I/O projects.

Papers in This Track

The Parallel I/O and Storage Technology track at Euro-Par 2000 contains six papers that address different aspects of the I/O problem:

1. "Towards a High-Performance and Robust Implementation of MPI-IO on top of GPFS," by Jean-Pierre Prost, Richard Treumann, Richard Hedges, Alice Koniges, and Alison White describes IBM's implementation of the MPI-IO standard for the GPFS file system.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1251–1252, 2000.
© Springer-Verlag Berlin Heidelberg 2000

2. "Design and Evaluation of a Compiler-Directed Collective I/O Technique," by Gokhan Memik, Mahmut T. Kandemir, and Alok Choudhary presents a compiler-directed collective I/O approach that detects opportunities for using collective I/O in a program and inserts the appropriate collective I/O calls.
3. "Effective File-I/O Bandwidth Benchmark," by Rolf Rabenseifner and Alice E. Koniges describes a benchmark designed to measure the effective I/O bandwidth achievable by applications on a given parallel machine and file system.
4. "Instant Image: Transitive and Cyclical Snapshots in Distributed Storage Volumes," by Prasenjit Sarkar presents an algorithm for handling snapshots of storage volumes in a distributed storage system.
5. "Scheduling Queries for Tape-Resident Data," by Sachin More and Alok Choudhary investigates issues in optimizing I/O time for a query whose data resides on an automated tertiary storage system containing multiple storage devices.
6. "Logging RAID – An Approach to Fast, Reliable, and Low-Cost Disk Arrays," by Y. Chen, W. Hsu, and H. Young presents a disk-array architecture that uses logging techniques to solve the small-write problem in parity-based disk arrays.

Towards a High-Performance Implementation of MPI-IO on Top of GPFS

Jean-Pierre Prost[1], Richard Treumann[2], Richard Hedges[3], Alice Koniges[3], and Alison White[2]

[1] IBM T.J. Watson Research Center, Route 134,Yorktown Heights, NY 10598
[2] IBM Enterprise Systems Group, 2455 South Road, Poughkeepsie, NY 12601
[3] Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore, CA 94550

Abstract. MPI-IO/GPFS is a prototype implementation of the I/O chapter of the Message Passing Interface (MPI) 2 standard. It uses the IBM General Parallel File System (GPFS) as the underlying file system. This paper describes the features of this prototype which support its high performance. The use of hints allows tailoring the use of the file system to the application needs.

1 Introduction

To provide users with a portable and efficient interface for parallel I/O, an I/O chapter was introduced in the Message Passing Interface 2 standard [3], based upon earlier collaborative work between researchers at the IBM T.J. Watson Research Center and the Nasa Ames National Laboratory [2].

Since approval of the MPI-2 standard, IBM has been working on both prototype and product implementations of MPI-IO for the IBM SP system, using the IBM General Parallel File System [4] as the underlying file system. This paper describes features of the prototype, referred to as MPI-IO/GPFS[1].

The use of GPFS as the underlying file system offers the potential for maximum performance through a tight interaction between MPI-IO and GPFS. GPFS is a high performance file system which presents a global view of files to every client node. It provides parallel access to disk data managed by server nodes through the IBM Virtual Shared Disk interface. GPFS provides coherent caching at the client and uses optimized prefetching techniques.

To avoid file block contention among tasks, MPI-IO uses data shipping. This technique binds each GPFS block to a single I/O agent, making it responsible for all accesses to this block. The GPFS blocks are bound to a set of I/O agents

[1] IBM product development work draws on the knowledge gained in this prototype project but features of the prototype discussed in this paper and features of eventual IBM products may differ. Any performance data contained in this paper was obtained using prototype software. Therefore, the results obtained with product software may vary significantly. Measurements quoted in this presentation may have been made on development-level systems. There is no guarantee that these measurements will be the same on generally-available systems. Actual results may vary.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1253–1262, 2000.
© Springer-Verlag Berlin Heidelberg 2000

with a round-robin striping scheme and MPI-IO/GPFS transfers data between the tasks where MPI-IO calls occur and the responsible agents as needed. MPI-IO/GPFS allows the user to define the stripe size. The stripe size also controls the amount of buffer space each I/O agent uses in each data access operation.

MPI-IO/GPFS is also a robust and user-friendly implementation. It prevents deadlocks when an error occurs only on a subset of the tasks participating in a collective I/O operation. In addition, errors which occur at the file system level can be traced on a per I/O agent basis through an optional error reporting feature. These robustness features are beyond the scope of this paper.

The paper is organized as follows. Section 2 details how data shipping is implemented in MPI-IO/GPFS. Section 3 presents performance measurements to demonstrate the benefit of using data shipping appropriately. Section 4 briefly describes features that are being implemented in the prototype in order to achieve a tighter integration between MPI-IO and GPFS. Section 5 presents some conclusions and suggests possible future research directions for the MPI-IO/GPFS prototype. All figures referenced in the text are gathered at the end of the paper.

2 MPI-IO/GPFS Features

The design foundation of MPI-IO/GPFS is the technique we call data shipping. To prevent conflicting access of GPFS file blocks by multiple tasks, normally residing on separate nodes, MPI-IO/GPFS binds each GPFS file block to a single I/O agent, which will be responsible for all accesses to the block. For an MPI-IO write call, the MPI task at which the call occurs ships to one or more I/O agents, a command with that agent's write assignment and the data to be written. The agents then perform their assigned file writes. For an MPI-IO read call, the task ships commands with I/O agent read assignments. The agents read the file as instructed and ship the data to the commanding tasks. The binding scheme implemented by MPI-IO/GPFS consists in assigning the GPFS blocks to the set of I/O agents according to a round-robin striping, illustrated in Figure 1. I/O agents are multi-threaded and are also responsible for combining data access requests issued by all participating tasks in a collective MPI-IO operation.

On a per file basis, the user can define the stripe size used in the allocation of GPFS blocks to I/O agents. The stripe size is the value of file hint IBM_io_buffer_size, which can be specified when the file is opened, or when the MPI_FILE_SET_INFO and MPI_FILE_SET_VIEW functions are called. It is possible for a program to change the stripe size of an opened file as long as no I/O operation is pending on the file. The stripe size also controls the amount of buffer space used by each I/O agent in data access operations, justifying its name. The stripe size given by the user is rounded up by MPI-IO/GPFS to an integral number of GPFS blocks, and its default size is the number of bytes contained in 16 GPFS blocks. GPFS block size defaults to 256K unless set to some other value by a system administrator when the GPFS file system is configured.

Finally, on a per file basis, the user can enable or disable MPI-IO data shipping by setting file hint IBM_largeblock_io to "false" or "true", respectively. MPI data shipping is enabled by default. When it is disabled, tasks issue read/write calls to GPFS directly. This saves the cost of MPI-IO data shipping but risks GPFS block ping-ponging among nodes if tasks located on distinct nodes contend for GPFS blocks in read-write or write-write sharing mode. In addition, collective data access operations are done independently. Therefore, we recommend to disable data shipping on a file only when accesses to the file are performed in large chunks or when tasks access disjoint large regions of the file. In such cases, MPI-IO coalescing of the I/O requests of a collective data access operation cannot provide benefit and GPFS block contention is not a concern.

It is worth commenting on the difficulty of describing the purpose of a file hint in terms which are meaningful to the user. The relationship between MPI calls to do I/O and the activity at the file system level is complex and largely opaque to the MPI programmer. This makes the choice of a meaningful name and value format for a file hint somewhat challenging. The name should be mnemonic and the set or range of values the user selects from should have a recognizable relationship to the user's understanding of her program's I/O behavior. The user should not be expected to understand the internals of the MPI-IO implementation. In the case of data shipping, the user can use two file hints which are called IBM_largeblock_io and IBM_io_buffer_size. These hints do not refer directly to MPI-IO/GPFS's data shipping mode. Instead, they allow the user to express the general I/O pattern of her application on a per file basis and how much buffer space should be made available to MPI-IO/GPFS to process each data access operation.

3 Performance Measurements

3.1 Benchmark Description

Let us first describe the two benchmarks used in our experimentation. Each is aimed at evaluating the benefit of the high performance features of MPI-IO/GPFS for a particular class of application I/O patterns. For all tests, the metric is read or write bandwidth expressed as Mbyte/second for the job as a whole. The benchmarks run with any number of tasks and provide every task with the same amount of I/O to do. In our tests, the file size scales with the number of tasks. To provide a consistent approach to measurement, in each benchmark, MPI_Barrier is called before beginning the timing, making each task's first call to the MPI read or write routine well synchronized. In MPI semantics, the return from an MPI write operation does not guarantee that data has been committed to disk. To ensure that the entire write time is counted, each test does an MPI_File_sync and MPI_Barrier before taking the end time stamp. In non-collective I/O it is possible that some tasks do I/O faster than others. Perhaps an argument could be made for averaging the times but we chose not to do that. The synchronizations ensure, in both collective and noncollective tests, we measure job time which we consider most meaningful.

Contiguous Benchmark In the contiguous benchmark, each task reads or writes sequentially a block of data from/to a single contiguous region of a pre-existing file. Region size, type of file access (read or write), and type of I/O operation (collective or noncollective) are parameters to the benchmark program. The number of tasks in the parallel job together with the region size determine the size of the file that is accessed.

Discontiguous Benchmark In the discontiguous benchmark, each task reads or writes an equal number of 1024-byte blocks, where the set of blocks that any one task reads or writes is scattered randomly across the entire file. Working together, the tasks tile the entire file without gaps or overlap. Parameters control file size, type of file access (read or write), type of I/O operation (collective or noncollective).

The benchmark program reads or writes the file region by region, using a two gigabyte region size. The region size is chosen as the maximum that could be mapped with an MPI_Datatype. At program start, each task is assigned a list of 1024-byte blocks to read or write to the file. At each step, the task creates an MPI_Datatype to map the blocks that belong to the current region into their proper file locations. This is accomplished by passing the MPI_Datatype that maps the blocks of the current region as a new filetype argument to MPI_File_set_view at each step.

3.2 Experimental Platform

All measurements were made on Lawrence Livermore National Laboratory ASCI Blue Pacific systems. These IBM RS6000/SP systems are based on 4-way nodes utilizing 332 Mhz 604e Power2 processors. Each of the nodes has 1.5 Gb of memory. The particular system hosting the experimentation includes 425 compute nodes and 56 I/O nodes. The GPFS configuration includes 38 VSD servers. GPFS page pools occupy 50 MB on each compute node. The software configuration includes AIX 4.3.2+, PSSP 3.1, and GPFS 1.2. A more detailed description of the file system configuration and GPFS performance statistics can be found elsewhere [5].

While it is preferable to make performance measurements on a dedicated system, the ASCI Blue systems are so heavily used that this was not possible. Thus, these measurements were carried out on systems in normal production operation. Despite this normal workload, efforts were made to make the measurements at times when other jobs were not contending for the GPFS file system. The peak performance on the contiguous benchmark with data shipping could be somewhat higher than our numbers, since it is affected by switch traffic and we are not in a dedicated environment. The peak performance we report for no data shipping is unlikely to change in a dedicated environment because our results are very reproducible. In order to obtain an estimate of the peak performance of the file system, we ran each of our experiments several times and then report the highest value, plotted in graphs collected at the end of the paper. We also use

the data obtained from several replications of the same runs in order to evaluate the impact of contention on the measurement results.

The contiguous and discontiguous benchmarks were run on 4, 8, 16, 32, 64, and 128 MPI tasks. Each MPI task had a node of the machine dedicated to it. Measurements for MPI-IO/GPFS are compared to the same tests using the ROMIO (MPICH) implementation [6,7] and to analogous tests utilizing a traditional POSIX I/O interface. By analogous, we mean that the POSIX tests use the same data layouts (across task memory and in the file) as the MPI-IO tests.

For all of the MPI-IO/GPFS measurements and for the POSIX version of the contiguous benchmark, I/O bandwidth was measured for seven replications of the same experiment. For the POSIX version of the discontiguous benchmark, the maximum achieved I/O bandwidths (< 10 MB/sec) ruled out multiple measurements. In this case, a single run of each experiment was performed. The maximum measured aggregated I/O bandwidth is reported for each combination of benchmark, read or write operation, MPI-IO implementation, and number of tasks.

3.3 Benchmark Results

The performance data is presented in graphs at the end of the paper, to provide a comparison of the aggregated bandwidths obtained with and without use of MPI-IO data shipping.

Contiguous Benchmark Results For the contiguous benchmark, described in Section 3.1, a region size of 333 MB was used. Each MPI task wrote a single region. In this situation, noncollective operations give the best performance. The data transfers are independent and can proceed in parallel with no coordination.

To illustrate the effect of using MPI-IO data shipping (controlled by the IBM_largeblock_io file hint), Figure 2 compares the results of running the contiguous benchmark with MPI-IO data shipping enabled (IBM_largeblock_io set to "false") to the same runs with MPI-IO data shipping disabled (IBM_largeblock_io set to "true"). For comparison, we also include results obtained with ROMIO and POSIX.

With data shipping disabled, the performance of MPI-IO/GPFS is strikingly similar to the ROMIO and POSIX versions of the benchmark. These three versions exhibit a saturation of GPFS's ability to process the data with saturation occurring between 32 and 64 MPI tasks.

For the contiguous benchmark with large file regions, there is no contention of tasks writing to the same GPFS block. Therefore, the setup logic and data movement involved in data shipping is purely overhead. For task numbers below 64, performance is less with data shipping enabled. The overhead of data shipping has an interesting effect for larger task numbers (64 and 128). We surmise that the overhead reduces the rate of queueing requests to GPFS providing a sort of flow control which leads to better performance with these large task numbers.

In Figure 3, results of the contiguous benchmark for read operations are displayed. Again with data shipping disabled, MPI-IO/GPFS performs like the

POSIX or ROMIO versions of the benchmark. In the case of reads, the data layout is such that no data shipping is needed for optimal performance. Flow control (or lack of it) is not an issue for the read tests. Enabling data shipping introduces additional overhead, and the read performance with data shipping enabled is consistently lower than with the other three versions of the benchmark.

The range of measured write bandwidths for the contiguous benchmark was observed to depend on the status of data shipping (controlled by the value of the IBM_largeblock_io hint). This variability of measured performance is particularly evident for larger numbers of tasks. In Figure 4, we produce the standard deviation, σ, of the aggregated write bandwidth for the two cases,

$$\sigma = \sqrt{\frac{n\Sigma x^2 - (\Sigma x)^2}{n(n-1)}}$$

where x is a single bandwidth measurement and n is the number of measurements. (We removed one data point from our sample that had a very low value potentially due to system problems.) The standard deviation with data shipping enabled exceeds the standard deviation with data shipping disabled for experiments with 32 or greater number of tasks. For 128 tasks, the standard deviation with data shipping is 35 times larger. The data is for MPI-IO/GPFS only, since the results for ROMIO and POSIX are very similar to those obtained with MPI-IO/GPFS and data shipping disabled.

A possible explanation for the dependence on data shipping, of the variability (expressed in terms of the standard deviation) of the measured bandwidth is the following. When data shipping is enabled, all data must be sent from the tasks to the I/O agents, and write calls are issued by the I/O agents. We notice an increasing variability in the bandwidth with an increasing number of tasks since there is more contention for switch access and because all tasks are trying to send data concurrently to each I/O agent. When data shipping is disabled, i.e., the IBM_largeblock_io is set to "true", there is no message passing occurring between the MPI tasks and the I/O agents during data access operations. Here, tasks are issuing I/O calls to GPFS directly. In this latter case, the variability in the measurements goes down to an almost insignificant level at larger numbers of tasks. The turnover in the standard deviation curve without data shipping occurs when the write rates begin to saturate because of the lack of flow control in GPFS 1.2. In summary, users on a production machine such as this one may see a large variability in their write performance for large numbers of tasks with data shipping, and significantly reduced variability in their write performance without data shipping.

Discontiguous Benchmark Results The discontiguous benchmark, described in Section 3.1, is most efficient with use of the optimized collective operations and data shipping enabled. The file size in this benchmark amounts to 333 MBytes per MPI task. Again, the POSIX and ROMIO versions of this benchmark are compared to MPI-IO/GPFS.

In Figure 5, results of the write performance tests for the discontiguous benchmark are presented. Initially, the most striking aspect of the plot is the dismal performance when using the straight POSIX interface. It is however not so surprising that many tasks writing 1KB blocks to a GPFS file system leads to very poor performance. Experimental results show that MPI-IO/GPFS with data shipping disabled leads to qualitatively similar performance.

In Figure 6, results of the read performance for the discontiguous benchmark are presented. Again, the performance of the POSIX version of the test shows poor performance as we would expect. MPI-IO/GPFS and ROMIO show good performance, achieving over 375 MB/sec and 250 MB/sec respectively.

ROMIO shows substantial optimization of the I/O in comparison to the POSIX results. The performance of ROMIO shows qualitatively different behaviors for the write and the read tests. Where performance of the read test is scaling well, performance of the write test peaks in the neighborhood of 32 tasks and then declines as the number of tasks increases.

We suspect that the additional load to GPFS of the read - modify - write cycle for data sieving in ROMIO has led to the same sort of flow control issues noted in the write performance of the contiguous test [6].

MPI-IO/GPFS shows the best write performance in the discontiguous benchmark reaching over 300 MB/sec and the best performance in the read test, reaching over 375 MB/sec. This demonstrates the benefit of using data shipping combined with collective operations, leading to good scalability for both read and write operations.

4 Work in Progress

We are currently experimenting with a tighter integration of MPI-IO with GPFS, through the use of prototyped GPFS directives and hints.

A new set of GPFS directives allow to set what is referred to as file partitioning mode at file open. When this mode is set, the file is partitioned into a number of large pieces. Each piece is accessed under the responsibility of a single node. In this mode, there is never any GPFS block shared between nodes so a single shared lock on the file is issued to each node. This eliminates any need to manage lock conflicts and greatly reduces the complexity and size of the state the GPFS lock manager maintains. MPI-IO/GPFS's data shipping feature is a natural match to GPFS file partitioning mode and can take advantage of the performance gain associated with it.

A new GPFS hint is being prototyped. It allows to prefetch selectively ranges of data contained in GPFS file blocks. MPI-IO/GPFS can use this hint at each I/O agent once it knows the GPFS blocks to be accessed for the current data access operation. If these blocks do not correspond to a sequential or strided sequential patterns, for which GPFS prefetching policy has been optimized, use of this hint has the potential to improve noticeably the I/O performance of the

MPI application. Because this hint is a part of the GPFS interface and is used by MPI-IO/GPFS, its exploitation is transparent to the MPI program making MPI-IO calls.

5 Conclusion

This paper illustrates how careful use of file hints can allow users to improve the performance of their MPI-IO applications on top of the IBM General Parallel File System. Data shipping is a clear winner when data is read-write or write-write shared among several tasks. Even though not illustrated in this paper, we also observed that increasing the stripe size and the I/O agent buffer size leads to better performance, provided the application can spare the buffer space. On the other hand, for applications in which tasks access either large pieces of data at once or disjoint regions of the file, enabling data shipping may reduce the performance.

We are currently investigating whether double buffering at the I/O agent can lead to increased performance. We are also examining whether data sieving, already used in ROMIO [6], could be beneficial to our prototype implementation. We are planning to study whether feedback from GPFS about the block hit ratio induced by its buffer cache replacement and prefetching policies or about the I/O request service time at the server side can be exploited by MPI-IO so that it can adapt its own prefetching policy or help GPFS control the I/O request flow to an overloaded server.

We also started to define synthetic benchmarks, which allow us to control the level of overlap between I/O and computation by using several user threads and adjusting the number of threads performing data access operations versus the number of computing threads. Through these synthetic benchmarks, we will also evaluate the impact of thread scheduling policies on the application performance.

6 Acknowledgements

All prototype and benchmark development was done by IBM personnel as part of ongoing research and development. Measurements and observations in the sections for Contiguous Benchmark Results and Discontiguous Benchmark Results have been provided by Lawrence Livermore National Laboratory. This portion of the work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

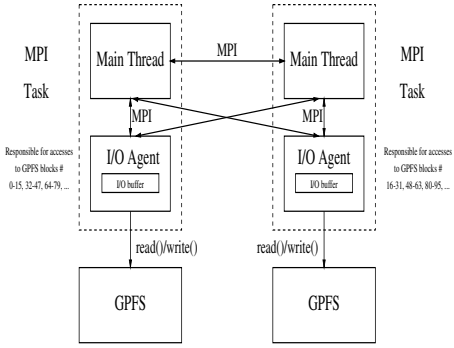


Fig. 1. GPFS block allocation used by MPI-IO/GPFS for a two task MPI job, in data shipping mode (using the default stripe size)

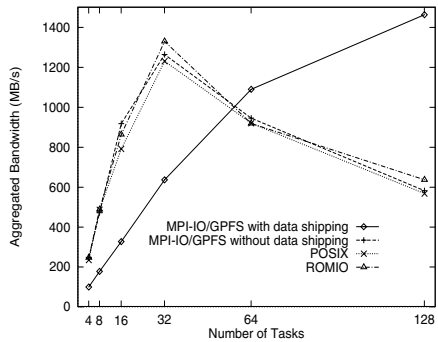


Fig. 2. Maximum bandwidths measured for write operations in the contiguous benchmark

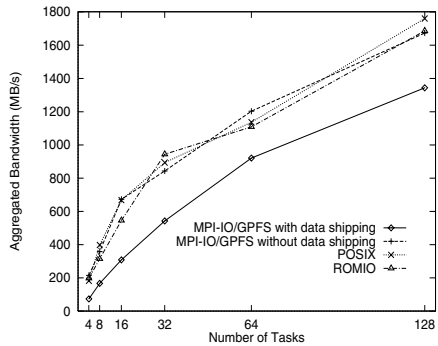


Fig. 3. Maximum bandwidths measured for read operations in the contiguous benchmark

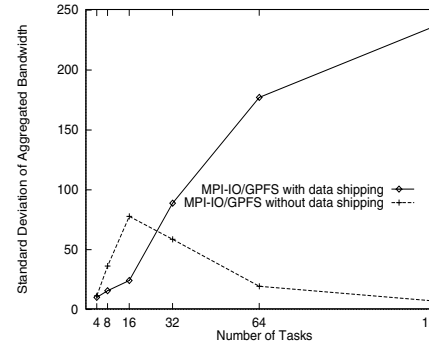


Fig. 4. Effect of data shipping on measured standard deviation for the contiguous write benchmark

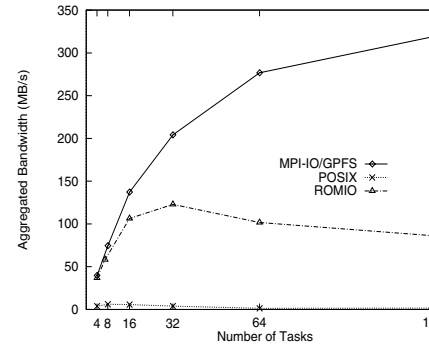


Fig. 5. Maximum bandwidths measured for write operations in the discontiguous benchmark

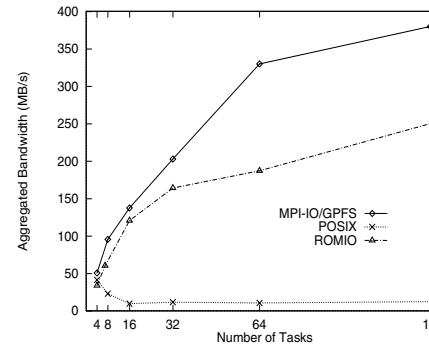


Fig. 6. Maximum bandwidths measured for read operations in the discontiguous benchmark

References

1. *www.llnl.gov/asci/*.
2. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, Chapter in *Input/Output in Parallel and Distributed Computer Systems*, Ravi Jain, John Werth, and James C. Browne, Eds., Kluwer Academic Publishers, June 1996, pp. 127–146.
3. Message Passing Interface Forum, *Mpi-2: A message-passing interface standard*, Standards Document 2.0, University of Tennessee, Knoxville, July 1997. *www.mpi-forum.org/docs/docs.html*.
4. *IBM General Parallel File System for AIX: Installation and Administration Guide*, IBM Document SA22-7278-02, October 1998.
   www.rs6000.ibm.com/resource/aix_resource/sp_books/gpfs/install_admin/install_admin_v1r2/gpfs1mst.html.
5. T. Jones, A. Koniges, and K. Yates, *Performance of the IBM General Parallel File System*, Proc. International Parallel and Distributed Processing Symposium, May 2000. Accepted.
6. R. Thakur, W. Gropp, and E. Lusk, *Data Sieving and Collective I/O in ROMIO*, Proc. 7th Symposium on the Frontiers of Massively Parallel Computation, February 1999, pp. 182–189.
7. R. Thakur, W. Gropp, and E. Lusk, *On Implementing MPI-IO Portably and with High Performance*, Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, May 1999, pp. 23–32.

Design and Evaluation of a Compiler-Directed Collective I/O Technique*

Gokhan Memik[1], Mahmut T. Kandemir[2], and Alok Choudhary[1]

[1] Department of Electrical and Computer Eng.
Northwestern University, Evanston IL 60208, USA
[2] Dept. of Computer Science and Engineering,
Pennsylvania State University, University Park PA 16802, USA

Abstract. Current approaches to parallel I/O demand extensive user effort to obtain acceptable performance. This is in part due to difficulties in understanding the characteristics of a wide variety of I/O devices and in part due to inherent complexity of I/O software. While parallel I/O systems provide users with environments where large datasets can be shared between parallel processors, the ultimate performance of I/O-intensive codes depends largely on the relation between data access patterns and storage patterns of data in files and on disks. Collective I/O is one of the most popular methods to access the data when the storage and access patterns do not match. In this strategy, each processor does I/O on behalf of other processors if doing so improves the overall performance. While it is generally accepted that collective I/O and its variants can bring impressive improvements as far as the I/O performance is concerned, it is difficult for the programmer to use collective I/O effectively. In this paper, we propose and evaluate a compiler-directed collective I/O approach which detects the opportunities for collective I/O and inserts the necessary I/O calls in the code automatically. An important characteristic of the approach is that instead of applying collective I/O indiscriminately, it uses collective I/O selectively, only in cases where independent parallel I/O would not be possible. We have conducted several experiments using an IBM SP-2 distributed-memory message-passing machine with 128 nodes. Our compiler directed collective I/O scheme was able to perform 18% better in average than an indiscriminate collective I/O scheme in our base configuration.

1 Introduction

Todays' parallel architectures comprise fast microprocessors, powerful network interfaces, and storage hierarchies that typically have multi-level caches, local and remote main memories, and secondary and tertiary storage devices. In going from upper levels of a storage hierarchy to lower levels, average access times

* This work is supported by the Department of Energy's Accelerated Strategic Computing Initiative (ASCI) program under a subcontract No W-7405-ENG-48 from Lawarence Livermore National Laboratories and by NSF CDA-9703228 and NSF ACI-9707074.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1263–1272, 2000.
© Springer-Verlag Berlin Heidelberg 2000

increase dramatically. Because of their cost effectiveness, magnetic disks have dominated the secondary storage market for the last several decades. Unfortunately, their access times have not kept pace with performance of the processors used in parallel architectures. Consequently, a large performance gap between secondary storage access times and processing unit speeds has emerged.

To address this imbalance, hardware designers focus on improving parallel I/O capabilities using multiple disks, I/O processors, and large bandwidth I/O busses [6]. An optimized I/O software can also play a major role in bridging this performance gap. In order to eliminate the difficulty in using a parallel file system directly, several research groups proposed high-level parallel I/O libraries and runtime systems that allow programmers to express access patterns of their codes using program-level data structures such as rectilinear array regions [4,13,5]. While all these software supports provide an invaluable help to boost the I/O performance in parallel architectures, it remains still programmer's responsibility to select appropriate I/O calls to use, to insert these calls in appropriate locations within the code, and to manage the data flow between parallel processors and parallel disks.

One of the most important optimizations in MPI-IO [5] is *collective I/O,* an optimization that allows each processor to do I/O on behalf of other processors [4]. This optimization has many variants [12,11,13]; the one used in this study is *two-phase I/O*. In this implementation, I/O is performed in *two phases*: an I/O phase and a communication phase. In the I/O phase, processors perform I/O in a way that is most beneficial from the storage layout point of view. In the second phase, they engage in a many-to-many type of communication to ensure that each piece of data arrives in its final destination. While collective I/O and its variants are very beneficial if used properly, almost all previous studies considered a *user-oriented* approach in applying collective I/O. For example, Thakur et al. suggest programmers to use collective I/O interfaces of MPI-IO instead of easy-to-use Unix-like interfaces [14]. Apart from determining the most suitable collective I/O routine and its corresponding parameters, this also requires, on the user part, analyzing access patterns of the code, detecting parallel I/O opportunities, and finally deciding a parallel I/O strategy.

In this paper, we propose and evaluate a *compiler-directed* collective I/O strategy whereby an optimizing compiler and MPI-IO cooperate to improve I/O performance of scientific codes. The compiler's responsibility in this work is to analyze the data access patterns of individual applications and determine suitable file storage patterns and I/O strategies. Our approach is *selective* because it activates collective I/O selectively, only when necessary. In other cases, it ensures that processors perform *independent parallel I/O,* which has almost the same I/O performance as collective I/O but without extra communication overhead.

The remainder of this paper is organized as follows. In the next section, we review collective I/O. In Section 3, we explain our compiler analyses to detect access patterns and suitable storage patterns for multidimensional datasets considering multiple, related applications together. In Section 4, we describe

our experimental framework, our benchmarks, and different code versions, and present our experimental results. In Section 5, we present our conclusions.

2 Collective I/O

In many I/O-intensive applications that access large, multidimensional, disk-resident datasets, the performance of I/O accesses depends largely on the layout of data in files (*storage pattern*) and distribution of data across processors (*access pattern*). In cases where these patterns are the same, potentially, each processor can perform *independent parallel I/O.* However, the term 'independent parallel I/O' might be misleading, as, depending on the I/O network bandwidth, the number of parallel disks available, and the data striping strategies employed by the parallel file system, two processors may experience a conflict in accessing different data pieces residing on the same disk [6]. What we mean by 'independent parallel I/O' instead is that the processors can read/write their portions of the dataset (dictated by the access pattern) using only a few I/O requests *in the code,* each for a large number consecutive data items in a file. These independent source-level I/O calls to files are broken up into several system-level calls to parallel disks. This last aspect, however, is architecture and operating system dependent and is not investigated in this paper. Note that, in independent parallel I/O, there is *no* interprocessor communication or synchronization during I/O.

In cases where storage and access patterns do not match, allowing each processor to perform independent I/O will cause processors to issue many I/O requests, each for a small amount of consecutive data. In this paper, an access pattern which is the same as the corresponding storage pattern is called a *conforming* access pattern. Collective I/O can improve the performance in non-conforming cases by first reading the dataset in question in a conforming (storage layout friendly) manner and then redistributing the data among the processors to obtain the target access pattern. Of course, in this case, the total data access cost should be computed as the sum of I/O cost and communication cost. The idea is that the communication cost is typically small as compared to I/O cost, meaning that the cost of accessing a dataset becomes almost independent from its storage pattern.

Consider Figure 1 that shows both independent parallel I/O and collective I/O for a four processor case using a single disk-resident two-dimensional dataset. In Figure 1(a), the storage pattern is row-major (each circle represents an array element and the arrows denote the linearized file layout of elements) and the access pattern is row-wise (i.e., each of the four processors accesses two full-rows of the dataset). Since the access pattern and the storage pattern match, each processor can perform independent parallel I/O without any need of communication or synchronization. Figure 1(b), on the other hand, shows the case where collective I/O is required. The reason is that in this figure the storage pattern is row-major and the access pattern does not match it. As explained earlier, the I/O is performed in two phases. In the first phase, each processor accesses the

data row-wise, as if this was the original access pattern), and in the second step, an all-to-all communication is performed between the processors and each data item is delivered to its final destination.

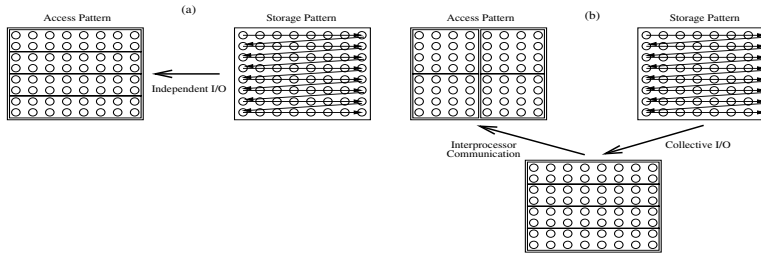


Fig. 1. (a) Independent parallel I/O and (b) Collective (two-phase) I/O.

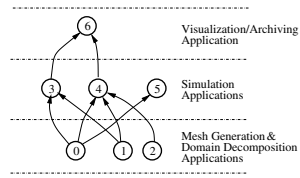


Fig. 2. Scientific working environment.

3 Compiler Analysis

Our approach to collective I/O utilizes a directed graph called *weighted communication graph* (WCG). Each node of a weighted communication graph is a *code block*, which can be defined as a program fragment during which we can keep the datasets in memory; however, between executions of code blocks, the datasets should be stored on disks. Depending on the applications, the datasets in question, and the available memory, a code block can be as small as a loop nest or can be as large as a full-scale application. An example for the latter is shown in Figure 2 that depicts a typical scenario from a scientific working environment. There is a directed edge, $e_{1,2}$, between two nodes, cd_1 and cd_2, of the WCG if and only if there exists at least a dataset that is produced (i.e., created and stored on disk) in cd_1 and used (i.e., read from disk) in cd_2. In such a case cd_1 is called *producer* and cd_2 is called *consumer*. The *weight* associated with $e_{1,2}$ (written as $w_{1,2}$) corresponds to total number of dynamic control-flow transitions between code blocks cd_1 and cd_2 (e.g., how many times cd_2 is run after cd_1 in a typical setup). Depending on the granularity of code blocks, these weights

can be calculated using profiling with typical input sets, can be approximated using weight estimation techniques, or can be entered by a user who observed the scientific working environment for a sufficiently long period of time.

3.1 Access Pattern Detection

Access patterns exhibited by each code block can be determined by considering individual loop nests that make up the code block. The crucial step in this process is taking into account the parallelization information [2]. Individual nests can either be parallelized explicitly by programmers using compiler directives [1,3], or can be parallelized automatically (without user intervention) as a result of intra-procedural and inter-procedural compiler analyses [2,8,9]. In either case, after the parallelization step, our approach determines the data regions (for a given dataset) accessed by each processor involved.

For each array reference in each loop nest, our compiler determines an access pattern. Afterwards, it utilizes a *conflict resolution scheme* to resolve intra-nest and inter-nest access pattern conflicts. To achieve a reasonable conflict resolution, we associate a *count* with each reference indicating (or approximating) the number of times that this reference is touched in a typical execution. In addition to that, for each access pattern that exists in the code block, we associate a *counter* that is initialized to zero and incremented by a count amount each time we encounter a reference with that access pattern. In this way, for a given array, we determine the most preferable (or most prevalent) access pattern (also called *representative access pattern*) and mark the code block with that information. Although, at first glance, it seems that, in a typical large-scale application, there will be a lot of conflicting patterns that would make compiler's job of favoring one of them over the others difficult, in reality, most scientific codes have a few preferable access patterns.

3.2 Storage Pattern Detection

Having determined an access pattern for each disk-resident dataset, the next step is to select a suitable storage pattern for each dataset in its producer code block. We have built a prototype tool to achieve this.[1]

For a given dataset, the tool takes the representative access patterns detected in the previous step by the compiler for each code block and runs a storage layout detection algorithm. Without loss of generality, in the following discussion, we focus only on a single dataset. The first step in our approach is to determine *producer-consumer subgraphs* (PCSs) of WCG for the dataset in question. A PCS for a dataset consists of a producer node and a number of consumer nodes that use the data produced by this producer. In the second step, we associate a count

[1] Note that building a separate tool is necessary only if the granularity of code blocks is a full-application. If, on the other hand, the granularity is a single nested-loop or a procedure, the functionality of this tool can be embedded within the compiler framework itself.

with each possible access pattern and initialize it to zero. Then, we traverse all the consumer nodes in turn and for each consumer node add its weight to the count of its access pattern. At the end of this step, for each access pattern, we obtain a count value. In the third step, we set the storage pattern in the producer node to the access pattern with the highest count. Note that, for a given dataset, we need to run the storage pattern detection algorithm multiple times, *one for each producer node* for this dataset.

The next step is to determine suitable I/O strategies for each consumer node. Let us again focus on a specific dataset. If the access pattern (for this dataset) of a consumer node is the same as the storage pattern in the producer node, we perform independent parallel I/O in this consumer node. Otherwise, that is, if the access and storage patterns are different, we perform collective I/O. We perform this step for each dataset and each PCS. Once the suitable I/O strategies have been determined, the compiler automatically inserts corresponding MPI-IO calls in each code block.

3.3 Discussion

Although, our approach is so far mainly discussed for a setting where individual code blocks correspond to individual applications, it is relatively easy to adapt it to different settings as well. If we consider each code block as a procedure in a given application, then a WCG can be processed using algorithms similar to those utilized in processing *weighted call graphs* [7], where each node represents a procedure and an edge between two nodes correspond to dynamic control-flow transitions (e.g., procedure calls and returns) between the procedures represented by these two nodes. In an out-of-core environment, on the other hand, each node may represent an individual loop nest and edges might represent dynamic control-flow between nests; in this case, the WCG is similar to a control-flow graph.

Another important issue that needs to be addressed is what to do (inside a code block) when we come across a reference whose access pattern is not the same as the representative (prevalent) access pattern for this code block. Recall that we assumed that, within a code block, we should be able to keep the datasets in memory. When the access pattern of an individual reference is different from the representative access pattern determined for a code block, we can re-shuffle the data in memory. This is not a correctness issue for shared-memory machines but it may cause performance degradation. In distributed-memory message-passing architectures, on the other hand, this data re-shuffling in memory is necessary to ensure the correct data accesses in subsequent computations.

4 Experiments

In this section, we first describe the experimental environment. Afterwards, we explain the setups for the experiments. Then, the results for the base configuration is given. Finally, we give results for different number of processors and data sizes.

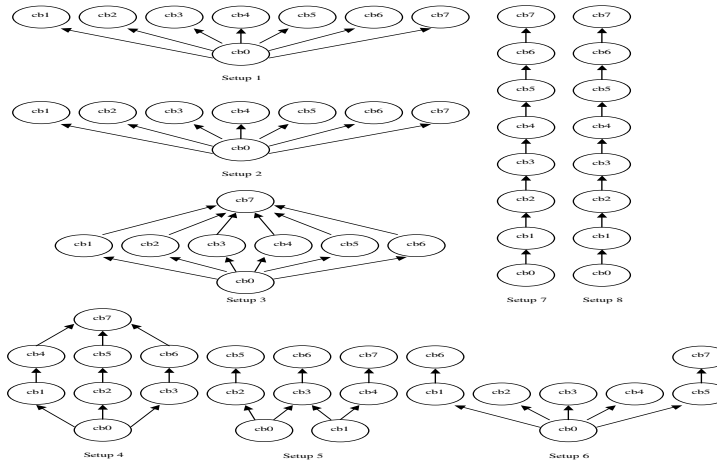


Fig. 3. Setups (communication graphs) used in the experiments.

4.1 Experimental Environment

We used the MPI-2 library and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this paper. The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. The compute nodes are RS/6000 Model 370 processors with 128 MB memory, whereas the I/O nodes are RS/6000 Model 970 processors with 256 MB memory. The nodes are connected via 100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI networks. Each I/O server has a 9 GB of storage space resulting in 72 GB of total disk space. The operating system on each node is AIX 4.2.1. PIOFS provides the parallel access to files. It distributes a file across multiple I/O server nodes.

4.2 Setups

To evaluate the possible improvements with our scheme, we have designed 8 different *setups* (*communication graphs*), each built up using 8 different benchmark codes in different ways. We have selected 4 benchmarks from Specfp (tomcatv (cb_0), vpenta (cb_1), btrix (cb_2), and mxm (cb_3)), 2 codes from the Perfect Club benchmark suite (tis (cb_4) and eflux (cb_5)), 1 from Nwchem suite (transpose (cb_6)) and a miscellaneous code (cholesky (cb_7)). The setups built from these benchmarks are given in Figure 3.[2]

[2] Although Setup 1 and Setup 2 look the same, they differ in the access and storage patterns they employ for different benchmarks. Similarly, Setup 7 and Setup 8 differ in their access and storage patterns. The details are omitted due to lack of space.

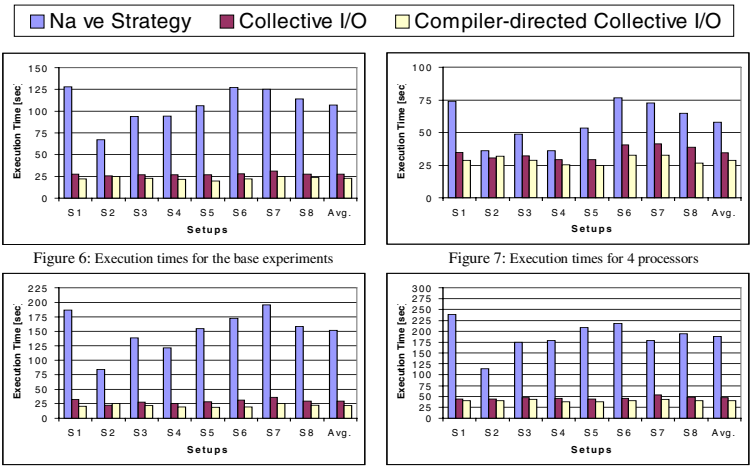


Figure 6: Execution times for the base experiments

Figure 7: Execution times for 4 processors

Figure 8: Execution times for 16 processors

Figure 9: Execution times for larger data size (Data size is doubled)

Three bars for each setup represents the following access strategies, from left to right: naïve strategy (no collective I/O), indiscriminate collective I/O, compiler directed collective I/O

4.3 Base Experiments

For the base experiments, we have executed the setup explained in Section 4.2 using 8 processors of the IBM SP-2. In our experiments we are comparing the execution times of three different code versions as explained below.

Version 1 In this version, each processor performs independent (non-collective) I/O regardless whether the access pattern and the storage pattern are the same or not. We call this version the *naive I/O strategy.*

Version 2 This version performs indiscriminate collective I/O.

Version 3 This is the strategy explained in this paper. The collective I/O is performed selectively, only if the access pattern and the storage pattern do not match. In all other cases, we perform independent parallel I/O.

In Figures 6 through 9, the left-most bar represents the total execution time of version 1, the middle bar represents the total execution time of version 2 and the right-most bar represents the total execution time of version 3.

The results for 8 processors are given in Figure 6. The average improvement over the indiscriminate collective I/O strategy is 18.01%. For setups 5 and 6, we are able to gain more than 21% over the version 2, which performs indiscriminate collective I/O. These two setups give the best results, because a change of the storage pattern effects the most applications. For example, when *cb*0 in Setup 1 changes its storage pattern, the weights of the favoring applications add up to 50% of the sum of all weights, whereas in Setup 6, the sum of weights of the favoring applications constitutes 60% of all the weights. So, there are 10% more favoring applications in Setup 6. Therefore, the improvement of Setup 6 is more than Setup 1 with our scheme.

Note that, both the indiscriminate and selective collective I/O strategies perform well compared to a naive strategy, which does not use collective I/O at

all. The improvement of indiscriminate collective I/O is 74.19% over the naive strategy, whereas our scheme brings 78.84% improvement.

4.4 Sensitivity Analysis

We have performed a second set of experiments to see how our strategy is affected by the number of processors and the data size. Figures 7 and 8 give the results for 4 and 16 processors, respectively. For 4 processors, our scheme brings 16.42% improvement over the indiscriminate collective I/O version, and 40.23% over the naive strategy. For 16 processors, on the other hand, our scheme brings an 26.27% improvement over the indiscriminate collective I/O version, and 80.74% over the naive strategy. As the number of processors increase, the improvement of our scheme increases, because with larger number of processors, the synchronization and communication costs increase.

Figure 9 gives the results for a larger data size. For this experiment, we have doubled the size of the input and/or output data of all the benchmarks. When the data size is increased, the synchronization overhead is reduced. Similarly, communication and I/O can be better overlapped because the I/O calls are longer, so the overall communication cost also reduces. These factors decrease the percentage improvements of our scheme, but, it still performs the best by far. It brings a 13.57% improvement over the indiscriminate collective I/O version, and 75.21% improvement over the naive strategy.

5 Conclusions

In this paper, we present and evaluate a compiler-directed collective I/O strategy. Collective I/O plays a major role in parallel I/O systems. Therefore, increasing its performance is very important for many of data-intensive parallel applications. By adopting a selective collective I/O strategy, we are able to bring significant amounts of improvements. In average, our scheme performs 18.01% better than an indiscriminate collective I/O strategy in our base configuration. The scheme performs better as the number of processors increases. Although the improvement decreases with the increased data size, the scheme is still able to perform more than 13% better than an indiscriminate collective I/O strategy.

The interface for parallel I/O systems are usually complex, and they are getting more complex, because the information required by the I/O calls increases. So, it becomes harder for an average user to detect the best possible I/O call for an application. Therefore, detecting the best possible storage and access patterns automatically is very useful for many programmers and increases the performance of I/O-intensive applications significantly.

References

1. A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Prog.,* 6(1):3–28, Spring 1997.

2. J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Computer Systems Lab., Stanford Univ., March 1997.
3. R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. Data-distribution support on distributed-shared memory multi-processors. In *Proc. Prog. Lang. Design and Implementation*, Las Vegas, NV, 1997.
4. A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. Passion: Parallel and scalable software for input-output. *NPAC Technical Report SCCS-636,* Sept 1994.
5. P. Corbett et al. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Par. and Dist. Sys.*, IPPS'95, Santa Barbara, CA, April 1995.
6. J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer,* March 1994.
7. N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proc. Micro-30,* Research Triangle Park, North Carolina, December 1–3, 1997.
8. M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.
9. M. W. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Inter-procedural analysis for parallelization. In *Proc. 8th International Workshop on Lang. and Comp. for Parallel Computers*, pages 61–80, Columbus, Ohio, August 1995.
10. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, March 1995.
11. D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation,* pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.
12. B. J. Nitzberg. *Collective Parallel I/O.* PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.
13. K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing'95,* December 1995.
14. R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived data types to improve I/O performance. In *Proc. of SC'98: High Performance Networking and Computing,* November 1998.

Effective File-I/O Bandwidth Benchmark

Rolf Rabenseifner[1] and Alice E. Koniges[2]

[1] High-Performance Computing-Center (HLRS), University of Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
rabenseifner@hlrs.de,
www.hlrs.de/people/rabenseifner/

[2] Lawrence Livermore National Laboratory, Livermore, CA 94550
koniges@llnl.gov,
www.rzg.mpg.de/∼ack

Abstract. The effective I/O bandwidth benchmark (b_eff_io) covers two goals: (1) to achieve a characteristic average number for the I/O bandwidth achievable with parallel MPI-I/O applications, and (2) to get detailed information about several access patterns and buffer lengths. The benchmark examines "first write", "rewrite" and "read" access, strided (individual and shared pointers) and segmented collective patterns on one file per application and non-collective access to one file per process. The number of parallel accessing processes is also varied and wellformed I/O is compared with non-wellformed. On systems, meeting the rule that the total memory can be written to disk in 10 minutes, the benchmark should not need more than 15 minutes for a first pass of all patterns. The benchmark is designed analogously to the effective bandwidth benchmark for message passing (b_eff) that characterizes the message passing capabilities of a system in a few minutes. First results of the b_eff_io benchmark are given for IBM SP, Cray T3E and NEC SX-5 systems and compared with existing benchmarks based on parallel Posix-I/O.

Keywords. MPI, File-I/O, Disk-I/O, Benchmark, Bandwidth.

1 Introduction

Most parallel I/O benchmarks and benchmarking studies characterize the hardware and file system performance limits [2,4,5,6]. Often, they focus on determining under which conditions the maximal file system performance can be reached on a specific platform. Such results can guide the user in choosing an optimal access pattern for a given machine and file system, but do not generally consider the needs of the application over the needs of the file system.

Our approach begins with consideration of the possible I/O requests of parallel applications. To formulate such I/O requests, the MPI Forum has standardized the MPI-I/O interface [7]. Major goals of this standardization are to express the user's needs and to allow optimal implementations of the MPI-I/O interface on all platforms [3,8,11,12]. Based on this background, the effective I/O bandwidth benchmark (b_eff_io) should measure different access patterns, report

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1273–1283, 2000.
© Springer-Verlag Berlin Heidelberg 2000

these detailed results, and should calculate an average I/O bandwidth value that characterizes the whole system. This goal is analogous to the Linpack value reported in TOP500 [16] that characterizes the computational speed of a system, and also to the effective bandwidth benchmark (b_eff), that characterizes the communication network of a distributed system [9,14,15].

A major difference between b_eff and b_eff_io is the magnitude of the bandwidth. On well-balanced systems in high performance computing we expect an I/O bandwidth which allows for writing or reading the total memory in approximately 10 **minutes**. For the communication bandwidth, the b_eff benchmark shows, that the total memory can be communicated in 3.2 **seconds** on a Cray T3E with 512 processors and in 13.6 seconds on a 24 processor Hitachi SR 8000. An I/O benchmark measures the bandwidth of data transfers between memory and disk. Such measurements are (1) highly influenced by buffering mechanisms of the underlying I/O middleware and filesystem details, and (2) high I/O bandwidth on disk requires, especially on striped filesystems, that a large amount of data must be transferred between such buffers and disk. Therefore a benchmark must ensure that a sufficient amount of data is transfered between disk and the application's memory. The communication benchmark b_eff can give detailed answers in about 2 minutes. Later we shall see that b_eff_io, our I/O counterpart, needs at least 15 minutes to get a first answer.

2 Multidimensional Benchmarking Space

Often, benchmark calculations sample only a small subspace of a multidimensional parameter space. One extreme example is to measure only one point, e.g., a communication bandwidth between two processors using a ping-pong communication pattern with 8 Mbyte messages, repeated 100 times. For I/O benchmarking, a huge number of parameters exist. We divide the parameters into 6 general categories. At the end of each category in the following list, a first hint about handling these aspects in b_eff_io is noted. The detailed definition of b_eff_io is given in section 4.

1. Application parameters are (a) the size of contiguous chunks in the memory, (b) the size of contiguous chunks on disk, which may be different in the case of scatter/gather access patterns, (c) the number of such contiguous chunks that are accessed with each call to a read or write routine, (d) the file size, (e) the distribution scheme, e.g., segmented or long strides, short strides, random or regular, or separate files for each node, and (f) whether or not the chunk size and alignment are wellformed, e.g., a power of two or a multiple of the striping unit. For b_eff_io, 36 different patterns are used to cover most of these aspects.
2. Usage aspects are (a) how many processes are used and (b) how many parallel processors and threads are used for each process. To keep these aspects outside of the benchmark, b_eff_io is defined as a maximum over these aspects and one must report the usage parameters used to achieve this maximum.

3. The major programming interface parameter is specification of which I/O interface is used: Posix I/O buffered or raw, special filesystem I/O of the vendor's filesystem, or MPI-I/O. In this benchmark, we use only MPI-I/O, because it should be a portable interface of an optimal implementation on top of Posix I/O or the special filesystem I/O.
4. MPI-I/O defines the following orthogonal aspects: (a) access methods, i.e., first writing of a file, rewriting or reading, (b) positioning method, i.e., explicit offsets, individual or shared file pointers, (c) coordination, i.e., accessing the file collectively by (all) processes or noncollectively, (d) synchronism, i.e., blocking or nonblocking. Additional aspects are: (e) whether or not the files are open *unique*, i.e., the file will not be concurrently opened by a different open call, and (f) which consistency is chosen for conflicting accesses, i.e., whether or not atomic mode is set. For b_eff_io there is no overlap of I/O and computation, therefore only blocking calls are used. Because there should not be a significant difference between the efficiency of using explicit offsets or individual file pointers, only the individual and shared file pointers are benchmarked. With regard to (e) and (f), *unique* and *nonatomic* are used.
5. Filesystem parameters are (a) which filesystem is used, (b) how many nodes or processors are used as I/O servers, (c) how much memory is used as bufferspace on each application node, (d) the disk block size, (e) the striping unit size, and (f) the number of parallel striping devices that are used. These aspects are also outside the scope of b_eff_io. The chosen filesystem, its parameters and any usage of non-default parameters must be reported.
6. Additional benchmarking aspects are (a) repetition factors, and (b) how to calculate b_eff_io, based on a subspace of the parameter space defined above using maximum, average, weighted average or logarithmic averages.

To reduce benchmarking time to an acceptable amount, one can normally only measure I/O performance at a few grid points of a 1-5 dimensional subspace. To analyze more than 5 aspects, usually more than one subspace is examined. Often, the common area of these subspaces is chosen as the intersection of the area of best results of the other subspaces. For example in [5], the subspace varying the number of servers is obtained with segmented access patterns, and with well-chosen block sizes and client:server ratios. Defining such optimal subspaces can be highly system-dependent and may therefore not be as appropriate for a b_eff_io designed for a variety of systems. For the design of b_eff_io, it is important to choose the grid points based more on general application needs than on optimal system behavior.

3 Criteria

The benchmark b_eff_io should characterize the I/O capabilities of the system. Should we use, therefore, only access patterns, that promise a maximum bandwidth? No, but there should be a good chance that an optimized implementation

type	l	L	U
0	1 MB	1 MB	0
	M_{PART}	$:=l$	4
	1 MB	2 MB	4
	1 MB	1 MB	4
	32 kB	1 MB	2
	1 kB	1 MB	2
	32 kB +8B	1 MB + 256B	2
	1 kB +8B	1 MB + 8kB	2
	1 MB +8B	1 MB + 8B	2

type	l	L	U
1	1 MB	$:=l$	0
	M_{PART}	$:=l$	4
	1 MB	$:=l$	2
	32 kB	$:=l$	1
	1 kB	$:=l$	1
	32 kB +8B	$:=l$	1
	1 kB +8B	$:=l$	1
	1 MB +8B	$:=l$	2

type	l	L	U
2	1 MB	$:=l$	0
	M_{PART}	$:=l$	2
	1 MB	$:=l$	2
	32 kB	$:=l$	1
	1 kB	$:=l$	1
	32 kB +8B	$:=l$	1
	1 kB +8B	$:=l$	1
	1 MB +8B	$:=l$	2
3/4	see type=2		
		$\sum U$	= 64

Table 1. The pattern details used in b_eff_io

of MPI-I/O should be able to achieve a high bandwidth. This means that we should measure patterns that can be recommended to application developers.

An important criterion is that the b_eff_io benchmark should only need about 10 to 15 minutes. For first measurements, it need not run on an empty system as long as concurrently running other applications do not use a significant part of the I/O bandwidth of the system. Normally, the full I/O bandwidth can be reached by using less than the total number of available processors or SMP nodes. In contrast, the communication benchmark b_eff should not require more than 2 minutes, but it must run on the whole system to compute the aggregate communication bandwidth. Based on the rule for well-balanced systems mentioned in the introduction and assuming that MPI-I/O will attain at least 50 percent of the hardware I/O bandwidth, we expect that a 10 minute b_eff_io run can write or read about 16 % of the total memory of the benchmarked system. For this estimate, we divide the total benchmark time into three intervals based on the following access methods: initial write, rewrite, and read. However, a first test on a T3E900-512 shows that based on the pattern-mix, only about the third of this theoretical value is transferred. Finally, as a third important criterion, we want to be able to compare different common access patterns.

4 Definition of the Effective I/O Bandwidth

The effective I/O bandwidth benchmark measures the following aspects:

- a *set of partitions*,
- the access methods *initial write*, *rewrite*, and *read*,
- the *pattern types* (see Fig. 1)
  (0) strided collective access, scattering large chunks in memory to/from disk,
  (1) strided collective access, but one read or write call per disk chunk,
  (2) noncollective access to one file per MPI process, i.e., on separated files,
  (3) same as (2), but the individual files are assembled to one segmented file,
  (4) same as (3), but the access to the segmented file is done with collective routines;
  for each pattern type, an individual file is used.

- the contiguous chunk size is chosen *wellformed*, i.e., as a power of 2, and *non-wellformed* by adding 8 bytes to the wellformed size,
- different chunk sizes, mainly 1 kB, 32 kB, 1 MB, and the maximum of 2 MB and 1/128 of the memory size of a node executing one MPI process.

The total list of patterns is shown in Tab. 1. The column "type" refers to the pattern type. The column "l" defines the contiguous chunks that are written from memory to disk and vice versa. The value M_{PART} is defined as *max(2 MB, memory of one node / 128)*. The column "L" defines the contiguous chunk in the memory. In case of pattern type (0), non-contiguous file views are used. If l is less than L, then in each MPI-I/O read/write call, the L bytes in memory are scattered/gathered to/from the portions of l bytes at the different locations on disk, see the left-most scenario in Fig. 1. In all other cases, the contiguous chunk handled by each call to MPI_Write or MPI_Read is equivalent in memory and on disk. This is denoted by ":=l" in the L column. U is a time unit.

Each pattern is benchmarked by repeating the pattern for a given amount of time. This time is given by the allowed time for a whole partition (e.g., $T = 10$ minutes) multiplied by $U/\sum U/3$, as given in the table. This time-driven approach allows one to limit the total execution time. For the pattern types (3) and (4) a fixed segment size must be computed before starting the pattern of these types. Therefore, the time-driven approach is substituted by a size-driven approach, and the repeating factors are initialized based on the measurements for types (0) to (2).

The b_eff_io value **of one partition** is defined as the sum of all transferred bytes divided by the total transfer time. If patterns do not need exactly the ideal allowed time, then the average is weighted by the unit U. At a minimum, 10 minutes must be used for benchmarking one partition. **The b_eff_io of a system** is defined as the maximum over any b_eff_io of a single partition of the system. This definition permits the user of the benchmark to freely choose the usage aspects and enlarge the total filesize as desired. The minimum filesize is given by the bandwidth for an initial write multiplied by 200 sec (= 10 minutes / 3 access methods). If a system complies with our rule that the total memory can be written in 10 minutes for each access pattern, then one third of the total memory is written by the complete benchmark, and in each single pattern with U=1, one 1/192 of the total memory is written. If all processors are used for this benchmark, then the amount written by each node is not very much, but a call to MPI_File_sync in each pattern may imply that the data is really written to disk. However this assumption is not valid on all systems. For example, on NEC SX systems, MPI_File_sync guarantees only the semantic stated in the MPI-2 standard. The data on the file must be visible to any other application, but the data can stay in a memory buffer controlled by the filesystem's software. Therefore the benchmark rule, that at least 10 minutes are used for one run, had to be modified for this system. In the current version we use for the SX-5 measurements, we require that the total amount of data written with the initial write-calls must be at least equal to the total amount of the memory of the

system. Thus, on the SX-5 we had to increase the scheduled benchmark time to T=30 minutes.

5 Comparing Systems Using b_eff_io

In this section, we present a detailed analysis of each run of b_eff_io on a partition. We test b_eff_io on three systems, the Cray T3E900-512 and SX-5Be/32M2 at HLRS/RUS in Stuttgart and an RS 6000/SP system at LLNL called "blue." On the T3E, we use the tmp-filesystem with 10 striped Raid-disks connected via a GigaRing for the benchmark. The peak-performance of the aggregated parallel bandwidth of this hardware configuration is about 300 MB/s. The LLNL results presented here are for an SP system with 336 SMP nodes each with four 332 MHz processors. Since the I/O performance on this system does not increase significantly with the number of processors on a given node performing I/O, all test results assume a single thread on a given node is doing the I/O. Thus, a 64 processor run means 64 nodes assigned to I/O, and no requested computation by the additional 64*3 processors. On the SP system, the data is written to the IBM General Parallel File System (GPFS) called blue.llnl.gov:/g/g1 which has 20 VSD I/O servers. Recent results for this system show a maximum read performance of approximately 950MB/sec for a 128 node job, and a maximum write performance of 690MB/sec for 64 nodes [5].[1] Note that these are the maximum values observed, and performance degrades when the access pattern and/or the node number is changed. The NEC SX-5 system has four striped RAID-3 arrays DS 1200, connected by fibre channel. The SFS filesystem parameters are: 4 MB cluster size (=block size), and if the size of an I/O request is less than 1 MB then a 2 GB filesystem-cache is used.

On both platforms, MPI-I/O is implemented with ROMIO but with different device drivers. On the T3E, we have modified the MPI Release mpt.1.3.0.2, by substituting the ROMIO/ADIO Unix filesystem driver routines for opening, writing and reading files. The Posix routines were substituted by the asynchronous counter part, directly followed by the the wait routine. This trick enables parallel disk access [13]. On the RS 6000/SP blue machine, GPFS is used underneath the MPICH version of MPI with ROMIO. On the SX-5, we use MPI/SX 10.1.

For each run of b_eff_io, the I/O bandwidth for each chunk size and pattern is reported in a table that can be plotted in the pictures shown in each row in Fig. 2. First, consider the first two rows of Fig. 2. They show the results of one benchmark on the SP and T3E systems, both scheduled to run $T = 10$ minutes, during which time other applications were running on the other processors of the systems. They demonstrate the main differences between both MPI and filesystem implementations. Based on the results in Fig. 3, which we discuss later, we decided to run the benchmark on the T3E on 32 processors and on the SP

[1] Upgrades to the AIX operating system and underlying GPFS software may have altered these performance numbers slightly between measurements in [5] and in the current work.

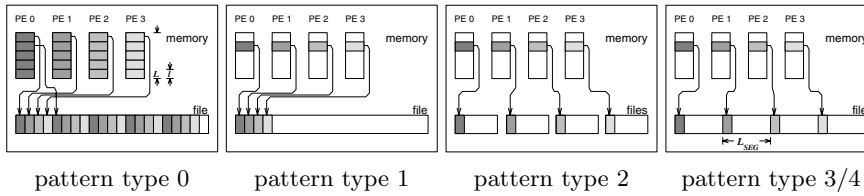
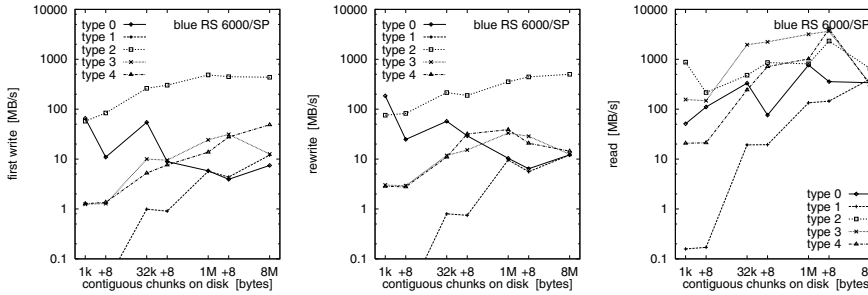
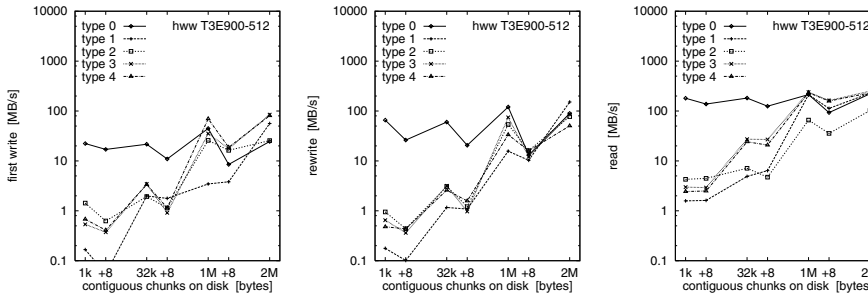


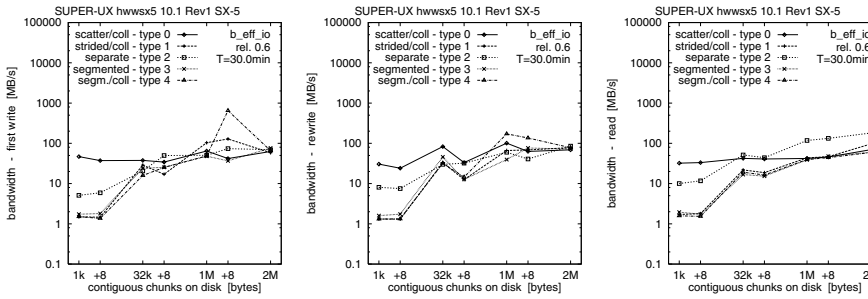
Fig. 1. Data transfer patterns used in b_eff_io. Each diagram shows the data transferred by **one** MPI-I/O write call.



(a) 128 PEs on the "blue" RS 6000/SP at LLNL, $T = 10$ min, b_eff_io = 311 MB/s



(b) 32 PEs on the T3E900-512 at HLRS, $T = 10$ min, b_eff_io = 71 MB/s



(c) 4 PEs on the SX-5Be/32M2 at HLRS, $T = 30$ min, b_eff_io = 60 MB/s

Fig. 2. Comparison of the results on T3E, SP and SX-5

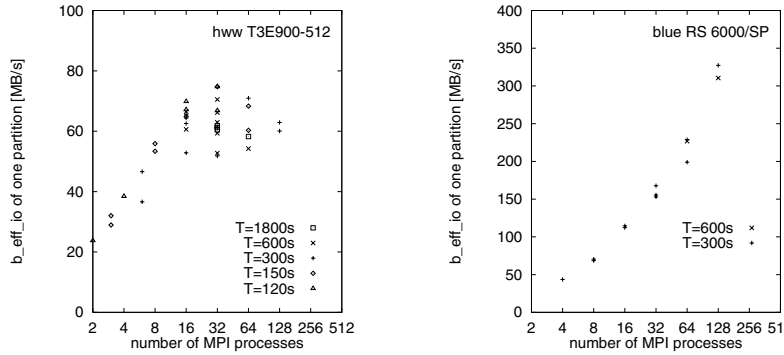


Fig. 3. Comparison b_eff_io of different numbers of processes on SP and T3E, measured partially without pattern type 3.

on 128 processors. The three diagrams in each row of Fig. 2 show the bandwidth achieved for the three different access methods: writing the file the first time, rewriting the same file, and reading it. On each diagram, the bandwidth is plotted on a logarithmic scale, separately for each pattern type and as a function of the chunk size. The chunk size on disk is shown on a pseudo-logarithmic scale. The points labeled "+8" are the non-wellformed counterparts of the power of two values. The maximum chunk size is different on both systems because the maximum chunk size was chosen proportional to the memory size per node to reflect the scaling up of applications on larger systems. On the SX-5, a reduced maximum chunk size was used.

Type 0 is a strided access, but the buffer used in each I/O-call is at least 1 MB. In the case of a chunk length less than 1 MB, the buffer contents must be scattered to different places in the file. On the T3E, this pattern type is optimal, except for chunks larger than 1 MB, where the initial write of segmented files is faster. When non-wellformed chunk sizes are used, there is a substantial drop in performance. Additional measurements show that this problem increases with the total amount of data written to disk. On the RS 6000/SP, other pattern types show higher bandwidth.

Type 1 writes the same data to disk, i.e., each process has the same logical fileview, but MPI-IO is called for each chunk separately. In the current benchmark, this test is done with individual filepointers, because the MPI-I/O ROMIO implementation on both systems does not have shared filepointers. By default, b_eff_io measures this pattern type with shared pointers when available. On both platforms, this pattern type results in essentially the worst bandwidth for most access methods and chunk sizes.

Type 2 is the writing winner on RS 6000/SP. Each process writes a separate file at the same time, i.e., parallel and independently. (We note that optimized vendor supplied MPI-IO implementations may do a better job with other pattern types.) Type 3 writes in the same pattern, but the files of all processes are

concatenated. To guarantee wellformed starting points for each process, the file-size of each process is rounded up to the next MByte. Type 4 writes in the same way as type 3, but the access is done collectively. On the T3E, we see that these three pattern types are consistently slow for small buffer sizes and consistently fast for large buffer sizes. In contrast on the RS 6000/SP, type 3 and 4 are about a factor[2] of **10–20** slower than type 2 for writing files. For reading files, the diagram cannot show the real speed for type 3 and 4 due to three effects: The repetition factor is only one for chunk sizes of 1 MB and more, the reading of the 8 MB chunk fills internal buffers, and currently, the b_eff_io does not perform a file sync operation before reading a pattern. Looking at the (non-weighted) average, we see that on the RS 6000/SP, reading the segmented files is a factor of **2.5** slower than reading individual files.

Finally on both systems, the read access is clearly faster than the write access. On the T3E, the read access is 5 times faster than "first write" and 2.7 faster than "rewrite". On the RS 6000/SP blue machine, the read access is 10 times faster than both types of write access. The measurements were done with b_eff_io Release 0.5 [10].

The last row of Fig. 2 shows the measurement on the SX-5. It had to be done with the longer schedule time of T = 30 minutes to assure that most of the I/O operations are done on real disks and not only in the filesystem's internal buffer space. The curves show still some hot spots that may be caused by pure memory copying. One can see that the scattering-pattern type 0 and the separate-file-pattern type 2 perform the best. There is little difference between wellformed and non-wellformed I/O. Write and read bandwidth are similar. For long chunk sizes, reading from separate files (pattern type 2) is faster than the gathering/strided accesses (type 0 and 1) and the segmented accesses (type 3 and 4).

Figure 3 shows the b_eff_io values for different partition sizes and different values of T, the time scheduled for benchmarking one partition. All measurements were taken in a non-dedicated mode. For the T3E, the maximum is reached at 32 application processes, with little variation from 8 to 128 processors. In general, an application only makes I/O requests for a small fraction of the compute time. On large systems, such as those at the High-Performance Computing Center at Stuttgart and the Computing Center at Lawrence Livermore National Laboratory, several applications are sharing the nodes, especially during prime time usage. In this situation, I/O capabilities would not be requested by a significant proportion of the CPU's at the same time. "Hero" runs, where one application ties up the entire machine for a single calculation are rarer and generally run during non-prime time. Such hero runs can require the full I/O performance by all processors at the same time. The right-most diagram shows that the RS 6000/SP fits more to this latter usage model. Note that GPFS on the SP's is configurable, i.e., number of I/O servers and other tunables, and the performance on any given SP/GPFS system depends on the configuration of that system.

[2] All factors in this section are computed, based on weighted averages using the time units U, if not stated otherwise.

Figure 3 also shows that on both systems, the results depend more on the I/O usage of the other concurrently running applications on the system than on the requested time T for each benchmark. Comparison of measurements with $T = 10$ and 30 minutes shows that the analysis reported in Fig. 2 may vary in details. For instance, the differences between wellformed and non-wellformed I/O is more notable with $T = 30$ minutes on the T3E.

Finally, we compare these results with other measurements. On the same RS 6000/SP, Posix read and write measurements ranging between 500 and 900 MB/s are measured [5]. [3] The b_eff_io result is 311 MB/s in the presented measurement. This means that the MPI application programmer has a real chance to get a significant part of the I/O capabilities of that system. On the T3E studied, the peak I/O-performance is about 300 MB/s. Thus the b_eff_io value of 71 MB/s shows that on average, only a quarter of the peak can be attained with normal MPI programming. We also note that the ROMIO implementation on the RS 6000/SP has not been optimized for the GPFS filesystem. Vendor implementations and future versions of ROMIO should show performance closer to peak.

In general, our results show that the b_eff_io benchmark is a very fast method to analyze the parallel I/O capabilities available for applications using the standardized MPI-I/O programming interface. The resulting b_eff_io value summarizes I/O capabilities of a system in one significant I/O bandwidth value.

6 Outlook

It is planned to use this benchmark to compare several additional systems. More investigation is necessary to solve problems arising from 32 bit integer limits and handling read buffers in combination with file sync operations. Although [1] stated, that "the majority of the request patterns are sequential", we should examine whether random access patterns can be included into the b_eff_io benchmark.

Acknowledgments

The authors would like to acknowledge their colleagues and all the people that supported this project with suggestions and helpful discussions. At HLRS, they would especially like to thank Karl Solchenbach and Rolf Hempel for productive discussions for the redesign of b_eff. At LLNL, they thank Kim Yates and Dave Fox. Work at LLNL was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

[3] Again we note that upgrades to the AIX operating system and underlying GPFS software may have slightly altered these performance numbers between measurements.

References

1. P. Crandall, R. Aydt, A. Chien, D. Reed, *Input-Output Characteristics of Scalable Parallel Applications*, In Proceedings of Supercomputing '95, ACM Press, Dec. 1995, `www.supercomp.org/sc95/proceedings/`.
2. Ulrich Detert, *High-Performance I/O on Cray T3E*, 40th Cray User Group Conference, June 1998.
3. Philip M. Dickens, *A Performance Study of Two-Phase I/O*, in D. Pritchard, J. Reeve (eds.), Proceedings of the 4th Internatinal Euro-Par Conference, Euro-Par'98, Parallel Processing, LNCS–1470, pages 959–965, Southampton, UK, 1998.
4. Peter W. Haas, *Scalability and Performance of Distributed I/O on Massively Parallel Processors*, 40th Cray User Group Conference, June 1998.
5. Terry Jones, Alice Koniges, R. Kim Yates, *Performance of the IBM General Parallel File System*, to be published in Proceedings of the International Parallel and Distributed Processing Symposium, May 2000. Also available as UCRL JC135828.
6. Kent Koeninger, *Performance Tips for GigaRing Disk I/O*, 40th Cray User Group Conference, June 1998.
7. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997, `www.mpi-forum.org`.
8. J.P. Prost, R. Treumann, R. Blackmore, C. Harman, R. Hedges, B. Jia, A. Koniges, A. White, *Towards a High-Performance and Robust Implementation of MPI-IO on top of GPFS*, Internal report.
9. Rolf Rabenseifner, *Effective Bandwidth (b_eff) Benchmark*, `www.hlrs.de/mpi/b_eff/`.
10. Rolf Rabenseifner, *Effective I/O Bandwidth (b_eff_io) Benchmark*, `www.hlrs.de/mpi/b_eff_io/`.
11. Rajeev Thakur, William Gropp, and Ewing Lusk, *On Implementing MPI-IO Portably and with High Performance*, in Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems, pages 23-32, May 1999.
12. Rajeev Thakur, Rusty Lusk, Bill Gropp, *ROMIO: A High-Performance, Portable MPI-IO Implementation*, `www.mcs.anl.gov/romio/`.
13. Rolf Rabenseifner, *Striped MPI-I/O with mpt.1.3.0.1*, `www.hlrs.de/mpi/mpi_t3e.html#StripedIO`.
14. Karl Solchenbach, *Benchmarking the Balance of Parallel Computers*, SPEC Workshop on Benchmarking Parallel and High-Performance Computing Systems, Wuppertal, Germany, Sept. 13, 1999.
15. Karl Solchenbach, Hans-Joachim Plum and Gero Ritzenhoefer, *Pallas Effective Bandwidth Benchmark – source code and sample results*, `ftp://ftp.pallas.de/pub/PALLAS/PMB/EFF_BW.tar.gz`.
16. Universities of Mannheim and Tennessee, *TOP500 Supercomputer Sites*, `www.top500.org`.

Instant Image: Transitive and Cyclical Snapshots in Distributed Storage Volumes

Prasenjit Sarkar

Storage Systems and Servers Division
IBM Almaden Research Center
San Jose, CA 95120
psarkar@almaden.ibm.com

Abstract. Snapshots are a useful mechanism to manage data, particularly for backups in high-availability storage systems. This paper presents the Instant Image algorithm for handling snapshots of storage volumes in a distributed storage system. The algorithm places no restrictions on the choice of storage volumes, allowing snapshot relationships between storage volumes that can be transitive and cyclical. In addition, in a distributed environment with snapshot relationships involving *n* storage subsystems, reads and writes to storage volumes involve only O(1) storage subsystems per read or write, thereby reducing messaging costs. Finally, the algorithm is not specific to storage systems and can be applied to other contexts requiring snapshots. A performance analysis indicates doubling of average read and write performance on a cluster of storage subsystems.

1. Introduction

Snapshots are a very useful mechanism to manage stored data. A snapshot of a continually changing storage volume is taken at a particular time, and captures the state of the storage volume at that time. Snapshots have been widely used in storage, filesystem and database architectures as they are very useful in various applications such as backup and testing[1,5,6,7]. For example, one problem of taking a backup of a storage volume is that the data in the volume changes while the volume is being backed up, making it difficult to get a consistent picture of the data. This is particularly true of terabyte storage volumes that may take hours to back up totally and requires administrators to block access to the storage volumes being backed up. Snapshots obviate this problem by creating a precise image of a storage volume at a particular time instant, and then taking a backup of the snapshot. The volume remains accessible to clients of the storage system and the administrator does not have to worry about the changes to the volume. This feature of snapshots endears it to the world of electronic commerce servers, where downtime is a luxury.

A parallel development fueled by the development of high-bandwidth networks is the development of distributed storage systems such as Petal[8] and xFS[2]. A distributed storage system consists of several storage subsystems (i.e., drive arrays) connected by a high-bandwidth network and presents a unified storage address space to the clients of the storage system. Distributed storage systems also allow for easy addition and removal of storage subsystems without any disruption of service. Last but

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1284-1291, 2000.
© Springer-Verlag Berlin Heidelberg 2000

importantly, distributed storage systems allow for a specialized storage system client to do backups and reduce the CPU utilizations of the remaining clients of the storage system. In a conventional storage system, performance is affected when a client backs up a storage volume while serving applications.

The instantaneous nature of snapshots gives rise to myriad relationships between storage volumes. For example, if an administrator takes a snapshot of a storage volume onto a second storage volume and the snapshot completes instantaneously, then nothing prevents the administrator from taking a snapshot of the second storage volume. Furthermore, there is no restriction on the destination of the second snapshot: in other words, the administrator may as well decide to take a snapshot of the second storage volume onto the first. This leads to both transitive and cyclical snapshot relationships between storage volumes. Existing systems like databases disallow transitive and cyclical relationships between snapshots to minimize the effect on transaction processing. However, in typical storage systems, such restrictions prevent useful applications from running on the snapshot. For example, a Web developer might want to test his application on a snapshot of an online database.

The implication of the emergence of distributed storage systems is that snapshots are no longer restricted to the same storage subsystem. For example, a distributed storage system may decide to allocate a given storage volume in any one of its component storage subsystems as a result of storage availability constraints. So any snapshot algorithm must take into the distributed nature of the problem and ensure that reads and writes to storage volumes do not suffer as a consequence of excessive messages being exchanged between different storage subsystems.

This paper presents the *Instant Image* algorithm for enabling snapshots in a storage system with the ability to handle transitive and cyclical snapshot relationships between storage volumes. The Instant Image algorithm places no limit on the number of snapshot relationships in the storage system and is independent of the actual physical layout of a storage volume. Furthermore, the algorithm ensures that in a distributed environment with n storage subsystems, every read or write to any storage volume does not involve more than O(1) storage subsystems independent of the number of snapshots in the storage system. In fact, performance analysis indicates a doubling of average read and write performance in a storage subsystem cluster. Finally, the algorithm is not specific to storage systems and can be applied without modification to other contexts that require snapshots.

The rest of the paper is organized as follows: Section 2 introduces definitions relevant to the Instant Image algorithm and is followed by Section 3 that describes the algorithm in detail. Relevant literature is described in Section 4, followed by conclusions in Section 5.

2. Definitions

A snapshot can be intuitively described as a “point-in-time” copy, where a snapshot of a storage volume behaves like a copy of the storage volume at the point of time the snapshot is taken. Thus, one key parameter of the snapshot is the time of operation

that is used to precisely define the actual contents of the snapshot. Following this, all reads and writes to the storage volume as well as the snapshot must honor the defined content of the snapshot. For example, if a block in a storage volume is written to after a snapshot is taken, then the original contents of the block must be transferred to the snapshot.

Formally, the Instant Image snapshot algorithm takes three parameters: a source storage volume *s,* a target storage volume *t* and a specified time instant *T.* Given the source storage volume *s* and target storage volume *t,* the Instant Image command at time *T* causes the target storage volume *t* to behave as a logical copy of the source storage volume *s* at time *T*. Following this, the storage volumes *s* and *t* are said to be part of the same snapshot relationship.

To maintain the semantics of a snapshot, reads and writes to *s* and *t* after time *T* are affected in the following manner:

- A read to *s* proceeds unhindered.
- A write to *t* also proceeds unhindered.
- A write to *s* results in the original data being copied over from *s* to *t* only if the data was not previously copied over from *s* to *t* and the data was not written to in *t*. Following this, the write is allowed to proceed.
- A read to *t* is redirected to *s* only if the data was not previously copied over from *s* to *t* and the data was not written to in *t*. Otherwise, the read proceeds unhindered.

There are no restrictions on the choice of *s* and *t.* In other words, the source and target storage volumes could be part of previous snapshot relationships with any other storage volume. The goal of the Instant Image algorithm is to take into account the effect of these previous snapshot relationships. Furthermore, the Instant Image algorithm is also independent of the actual physical layout of both the storage volumes involved in the snapshot relationship.

Snapshots require that the target storage volume be able to store data equal to the size of the source storage volume. If the entire source storage volume is written to, all the original data in the source storage volume must be copied over to the target storage volume. However, the Instant Image algorithm is independent of the exact mechanism by which this is achieved. In certain implementations, the source and target storage volumes share the underlying physical storage until writes to either of the volumes necessitates distinct physical storage[Hitz94]. In other implementations, the source and target storage volumes do not share any physical storage.

The Instant Image algorithm aims to achieve the following properties:

- *Transitive Relationships*: Given three storage volumes *A, B, C,* it is possible to specify a snapshot relationship with *A* as the source and *B* as the target, and afterwards specify another snapshot relationship with *B* as the source and *C* as the target.
- *Multiple Targets:* It is possible for a storage volume to be the source for many snapshot relationships. For example, given storage volumes *A, B, C, D,* the

following chronological sequence of snapshot relationships are valid: (i) *A* as the source and *B* as the target; (ii) *A* as the source and *C* as the target; (iii) *A* as the source and *D* as the target.

- *Cyclical Relationships*: It is possible to specify multiple snapshot relationships that are cyclical in nature. For example, given three storage volumes *A, B, C,* it is possible to specify the following snapshot relationships in chronological sequence: (i) *A* as the source and *B* as the target; (ii) *B* as the source and *C* as the target; (iii) *C* as the source and *A* as the target.
- *Distributed Relationships:* The Instant Image algorithm has efficient support for distributed relationships. In particular, given storage volumes involved in snapshot relationships (no restriction on either the nature or number) over *n* storage subsystems, the algorithm ensures that reads and writes to storage volumes involve only O(1) storage subsystems per read or write. This minimizes the number of request redirections and data copies that have to be performed in the storage system.

Finally, there is no limit to the number of concurrent snapshot relationships as long as there are enough physical resources (i.e., memory, disk space) to support the relationships.

3. Algorithm

The Instant Image algorithm assumes an array of storage volumes in a storage system consisting of *n* storage subsystems, with no restrictions on the distributions of the storage volumes over the storage subsystems. The storage volumes may be part of any number of previous snapshot relationships. The Instant Image operation is directed to a source data volume *s* and a target data volume *t* at time *T*.

In the remainder of the section, the metadata required for the Instant Image is described.

3.1 Metadata

Each storage volume has metadata associated with the volume to assist in the implementation of the Instant Image algorithm. Each storage volume is logically divided into *m* equal-sized segments, where *m* is proportional to the size of the storage volume. The size of a segment is fixed by the designer of the Instant Image algorithm and assumed to be constant across the storage system. It must be noted here that the algorithm is independent of the chosen segment size.

For every storage volume, there are two main metadata structures associated with the volume. For a storage volume *i*, these are Source(*i*)[1..*m*] and Target(*i*)[1..*m*], where *m*, as defined before, is maximum number of segments in the volume. The meaning of the structures is as follows:

- Source(*i*)[*j*] = *k* implies that the *j*th segment of storage volume *i* is physically located in the *j*th segment of storage volume *k*. Thus, the source of a storage

volume segment provides a pointer to the actual physical location of the segment. If $k = i$, it means that the segment is located within the volume itself.

- Target(i)[j] = $\{k_s\}$ means that for every storage volume k_s in the set defined by the Target field, the jth segment of k_s is located in the jth segment of storage volume i. This means that when a storage volume segment is written, the original contents of the segment must be written to the locations indicated by the Target field. If the Target field set is empty, it indicates that the original contents of the segment can be discarded.

When a storage volume i is created, then for each segments j in the volume, Source(i)[j] = i and Target(i)[j] = {}.

The relation between the Target and Source metadata structures is not symmetric. In other words, if Source(i)[j] = k, it does not necessarily mean that i is in Target(k)[j].

For algorithmic needs, a third auxiliary metadata structure to mirror the Target field is defined. This metadata structure is defined as Previous(i)[1..m], where m is the maximum number of segments in storage volume i. The meaning of this structure is simply: Previous(i)[j] = k <=> i is in Target(k)[j].

Finally, every storage subsystem keeps a table that maps a given storage volume to the corresponding storage subsystem where the volume is located.

3.2 Algorithm

Establishment. When an Instant Image operation is invoked with s as the source data volume and t as the target data volume, a snapshot relationship is said to be established between s and t. There are many choices in creating a snapshot relationship. For example, as is shown below, some choices degrade read performance, while others degrade write performance. The Instant Image algorithm establishes snapshot relationships in a manner such that both read and write performance is optimized.

The first part of the algorithm updates the Source metadata structure for the target storage volume for the snapshot. One alternative is to set Source(t)[j] = s for each segment j in t, such that requests for data in t get redirected to s. The disadvantage of the scheme is that it degrades read performance in situations where a segment read request to t gets redirected multiple times over the network before reaching the storage volume where the segment is physically located.

In the Instant Image algorithm, the Source metadata structure in the target storage volume is updated such that Source(t)[j] points to the storage volume which actual location of the data, so that at most one redirection is required. Thus, in an environment with multiple snapshot relationships involving n storage subsystems, a read for a segment would involve at most O(1) storage subsystems. In contrast, the alternative approach could involve O(n) storage subsystems in a similar situation.

Consequently, for all segments j in the target data volume t, we set:

$$\text{Source}(t)[j] = \text{Source}(s)[j] \qquad \text{(A)}$$

The second part of the establishment operation updates the Target metadata structure. One alternative is to mirror the Source metadata structure update in the previous step. Here, if after the Source metadata structure update, Source(t)[j] = u for a particular segment j, then we would add t to the set defined by Target(u)[j]. The disadvantage of this mechanism is that the cardinality of the set defined by the Target field for a segment can be high. Consequently, it affects write performance in situations where the original data for a segment must be written over the network to multiple storage volumes before a write for the segment is allowed to proceed.

In the Instant Image algorithm, the cardinality of the set defined by the Target field is restricted to one so as to improve write performance. The aim is to make sure that the original segment needs to be written only once before the write for the segment is allowed to proceed. Thus, instead of adding a storage volume to the set defined by the Target field for a segment, we create a linear chain of storage volumes, where the set defined by the Target field in a storage volume contains only the identity of the succeeding storage volume in the chain. Finally, the Previous metadata field for the segment is also updated to reflect the change to the Target field. This is achieved as follows:

var r = s
while (Target(r)[j] != {}) r = singleton element of Target(r)[j]
Add t to Target(r)[j]
Previous(t)[j] = r (B)

Once the linear chain is established, a write to a storage volume segment is allowed to proceed immediately after the original contents of the segment have been written to the succeeding storage volume in the chain. This means that the write to the original storage volume segment is delayed by *at most* one other write to the succeeding storage volume. The succeeding storage volume then writes the original data to its successor without holding up the original write to the storage volume segment. The sequence of writing to the successor continues till the end of the chain is reached. Thus, in an environment of n storage subsystems, a write to a segment involves at most O(1) storage subsystems. In contrast, the alternative could involve O(n) storage subsystems in the worst case where a write to a storage volume segment would be delayed by O(n) consecutive writes.

One corollary action that is done during the creation of the chain of storage volumes is the detection of cycles in the chain. Such cycles are eliminated by undoing the update to the Target and Previous fields to give precedence to existing snapshot relationships.

As is evident from the above, the Instant Image algorithm increases the complexity of establishment to optimize read and write performance. This is justified because reads and writes to storage volumes are expected to be more common than the establishment operations.

Consistency. The Instant Image algorithm must also update the Source and Target metadata structures when a segment is copied over from a source storage volume to target storage volume, or when a segment is written to in a target storage volume.

This update is necessary to maintain the semantics of a snapshot relationship (defined in Section 2).

Assume that an access to a storage volume causes either a segment *j* to be copied over to storage volume *t* or be overwritten in storage volume *t*. First, the Source metadata structure is updated in *t* to indicate that the segment j is located in the volume itself:

$$\text{Source}(t)[j] = t \tag{C}$$

The next step is to update the Target and Previous metadata structures to remove *t* from the linear chain of storage volumes:

$$\begin{aligned} &\text{var } p = \text{Previous}(t)[j],\ g = \text{Target}(t)[j] \\ &\text{Previous}(g)[\text{j}] = \text{Previous}(t)[j] \\ &\text{Target}(p)[j] = \text{Target}(t)[j] \end{aligned} \tag{D}$$

A prototype of the Instant Image algorithm was implemented in an IBM ServRAID III storage controller with a 40 MHz 403GCX processor and 32 MB of memory.

4. Related Work

Storage system snapshots are "dumb" snapshots of a storage volume in that the contents of the storage volume are not interpreted[7]. In contrast to storage system snapshots, file system and database snapshots are more intelligent and interpret the contents of the storage system to allow finer-grained snapshots. Examples of file systems with good snapshot features are the Echo file system[5] and the WAFL system[6]. Chervenak et al. provide a good survey of literature pertaining to this field[4]. Databases also provide snapshot features for users to query various epochs of the same database, and use various storage and algorithmic techniques to optimize query processing time against these snapshots[1,9]. However, a common feature that separates the snapshot strategy presented here is that the Instant Image snapshot algorithm allows normal access (including reads and writes) to snapshot volumes, allowing snapshot relationships that are both transitive and cyclical.

5. Conclusions and Future Work

This paper presents the Instant Image algorithm for providing snapshot support in a storage environment. The Instant Image algorithm can not only handle transitive and cyclical snapshot relationships, but also optimizes read and write message traffic in a distributed environment. This was achieved by considering various alternatives to establish a snapshot relationship, and then selecting the best approach that involved only O(1) storage subsystems per read or write in an environment of storage volumes spread over *n* storage subsystems.

References

1. Adiba, M., Lindsay, B.: Database Snapshots. Proceedings of VLDB (1980) 86-91.
2. Anderson, T., Dahlin, M., Neefe, J., Patterson, D., Roselli, D., Wang, R.: Serverless Network File Systems. ACM TOCS, Vol 14(1) (1996) 41-79.
3. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM TOCS, Vol 3(1) (1985) 63-75.
4. Chervenak, A., Velanki, V., Kurmas, Z.: Protecting File Systems: A survey of backup techniques. Proceedings of the Joint NASA-IEEE Mass Storage Conference (1998).
5. Chutani, S., Anderson, O., Kazar, M., Leverett, B., Mason, A., Sidebotham, R.: The Episode File System. Proceedings of the Winter Usenix Technical Conference (1992) 43-60.
6. Hitz, D., Lau, J., Malcom, M.: File system design for a File Server Appliance. Proceedings of Winter Usenix Technical Conference (1994) 235-245.
7. Hutchinson, N., Manley, S., Federwisch, M., Harris, G., Hitz, D., Kleinman, S., O'Malley, S.: Proceedings of OSDI Symposium. (1999) 239-249.
8. Lee, E., Thekkath, C.: Petal: Distributed Virtual Disks. Proceedings of ASPLOS V (1996) 84-92.
9. Lindsay, B., Haas, L., Mohan, C., Pirahesh, H., Wilms, P.: A Snapshot Differential Referesh Algorithm. Proceedings of SIGMOD (1986) 53-60.
10. Sato, Y., Inoue, M., Masuzawa, T., Fujiwara, H.: A Snapshot Algorithm for Distributed Mobile Systems. Proceedings of ICDCS (1996) 734-743.
11. Spezialetti, M., Kearns, P.: Efficient Distributed Snapshots. Proceedings of ICDCS (1986) 53-60.
12. Venkatesan, S.: Message-optimal Incremental Snapshots. Proceedings of ICDCS (1989) 382-288.

Scheduling Queries for Tape-Resident Data*

Sachin More and Alok Choudhary

Department of Electrical and Computer Engineering
Northwestern University

Abstract. Tertiary storage systems are used when secondary storage can not satisfy the data storage requirements and/or it is a more cost effective option. The new application domains require on-demand retrieval of data from these devices. This paper investigates issues in optimizing I/O time for a query whose data resides on automated tertiary storage containing multiple storage devices.

1 Introduction

Tertiary storage systems are employed in cases where secondary storage can not satisfy the data storage requirements or tertiary storage is a more cost effective option [8]. NASA's Earth Observing System (EOS) Data and Information System (EOSDIS) is an example of the former [4, 13]. The latter case, can be found in data warehousing applications. Inmon [10] shows that substantial monetary savings can be achieved using a hierarchical data storage containing comparatively small amount of secondary storage and vast amounts of tertiary storage without sacrificing performance.

Tertiary storage devices have traditionally been used as archival storage. The new application domains require on-demand retrieval of data from these devices [9]. While data archiving applications access large chunks of contiguous data, these new applications access data that is scattered on multiple media. Hence correct scheduling of data retrieval requests becomes important. For example, I/O time for a query that accesses data from two different media using a single robotic arm and two tape drives is minimized when the tape that needs more time to read is loaded first.

Many of the application domains that use tertiary storage access *multidimensional datasets* [2]. In a multidimensional dataset, each data item occupies a unique position in a n-dimensional hyper-space. A query selects a subset of the data items by selecting a subset of the domain in each dimension. Given the wide variety of expected queries [18], it is not possible to store data accessed by each query contiguously without high amount of data replication. Hence, a query on a multidimensional dataset stored on tertiary storage, accesses data from multiple media [15]. Time needed to read the query data from a media depends on the amount of data and location of the data inside the media. When using

* This work is supported by DOE ASCI Alliance program under a contract from Lawrence Livermore National Labs B347875.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1292–1301, 2000.
© Springer-Verlag Berlin Heidelberg 2000

automated tertiary storage, the total I/O time for the query is also influenced by the order in which media are accessed.

Various issues in tertiary storage management have been addressed by the database community. Carey et. al. [1] evaluate issues in extending database technology for storing/accessing data on tertiary storage. Stonebraker [23] proposes a database architecture that uses hierarchical storage. Livny et. al. [17] and Sarawagi [20, 21, 22] examine issues in query processing when data resides on tertiary storage. Data striping on tertiary storage has been evaluated in [3, 5]. Tertiary storage space organization issues are addressed in [2, 6]. This paper investigates issues in optimizing I/O time for a query whose data resides on automated tertiary storage containing multiple storage devices. We model the problem as a limited storage parallel two-machine flow-shop scheduling problem with additional constraints. Given a query, we establish an upper bound on the number of storage devices for an optimal I/O schedule. For queries that access small amounts of data from multiple media, we derive an optimal schedule analytically. For queries that access large amount of data we derive a heuristics-based scheduling algorithm using analytically proven results.

The rest of this paper is organized as follows. Section 2 introduces the problem. Sections 3, 4 and 5 analyze the problem and provide theoretical results for the problem. Section 6 discusses important practical considerations and presents performance evaluation of our approach. Conclusions are presented in Section 7.

2 Background

The system model consists of $A \geq 1$ robotic arms and $T > 1$ tape drives. A query needs data from n tapes. Reading data from a tape consists of the following set of operations: rewinding currently loaded tape; ejecting the tape; putting it back; fetching the tape to be read; loading the tape; searching and reading data inside the tape. Putting back a tape and fetching a new one are handled by the robotic arm and the rest of the actions are carried out by the tape drive. The time to do the arm operations is denoted by t_A (which we assume to be same for all tapes [6]) and time to do the drive operations is denoted by t_D.

Given a set of tapes and blocks from each tape that need to be accessed by a query, find the order in which the tapes should be read to minimize the total I/O time. The problem is cast as a special case of two-machine flow-shop scheduling problem [19]. The arm operations denote the first machine and the drive operations denote the second machine. There are n jobs to be scheduled. The optimality criteria is *makespan*, total execution time of the schedule. The distinctive features of our problem (in contrast to the traditional two-machine flow-shop scheduling problem) are:

- *Multiple instances of machines* More common system configurations have a single robotic arms servicing a number of tape drives. In this paper, we consider the case where there is one instance of the first machine and multiple instances of the second machine.

- *Buffer bound* $= T$ At the most T jobs can be in the shop simultaneously. The robotic arm can not load more tapes while all drives are busy accessing tapes loaded in them and must remain idle.
- If job i starts at time s_i on the first machine then it must be scheduled so that it finishes by $s_i + t_A + t_{D_i}$ on the second machine. The second machine is idle for time t_A before a job can be scheduled on it. This accommodates for the behavior of a tertiary storage system where a drive is empty while the robotic arm is loading the next tape. The case where $A = T = 1$ (a single tape drive serviced by a single robotic arm) is uninteresting under this condition. In the rest of this paper we assume $T > 1$.
- Practical considerations prevent use of scheduling algorithms that compare t_A and t_{D_i} values. The value of t_{D_i} can not be predicted correctly unless a very accurate analytical model of the tape drive is available. For example, Johnson's algorithm [12], which is optimal for traditional two-machine flow-shop scheduling, performs such a comparison.

3 Workload Characterization

A job is characterized by a k value of the job. For any job i, its k value is governed by the inequality $((k-1) \times t_A) < t_{D_i} \leq (k \times t_A)$. The k value of a workload is defined as $((k-1) \times t_A) < max_i(t_{D_i}) \leq (k \times t_A)$ where $0 \leq i < n$.

Theorem 1. [14] *For $A = 1$, if $(k-1) \times t_A < max_i(t_{D_i}) \leq k \times t_A$, then increasing number of instances of the second machine beyond $min(n, k+1)$ does not improve the makespan of any schedule.*

The above result provides an interesting insight to the problem. Given a workload, it tells us when the first machine is the bottleneck and when it is not. Given the system configuration, makespans of workloads with k values less than or equal to T will be constrained by the first machine, that is idle times can be introduced on the second machine because the first machine is always busy. The jobs in these kind of workloads have their execution time on the second machine bounded above by $T \times t_A$. Jobs in these workloads are *small jobs*. Execution time on the second machine for a *large job* is more than $T \times t_A$.

4 Workloads Consisting of Small Jobs

A workload containing small jobs represents a situation where after loading a media in a drive, the drive finishes reading data off that media before the robotic arm can finish loading media in other drives. In such a situation, we find that the robotic arm is busy all the time (except at the end when there are no more media to access) irrespective of the order in which the media are loaded.

Theorem 2. [14] *$\forall i$, if $t_{D_i} \leq (T-1) \times t_A$ and $A = 1$, then longest-t_D-first (LtF) schedule is optimal.*

5 Workloads Consisting of Big Jobs

For queries that access large amount of data from each tape (where $t_{D_i} > (T - 1) \times t_A$), the first machine is not a bottleneck. This leads one to believe that eliminating idle times on the second machine will lead to an optimal schedule.

Proposition 3. [14] *$\forall i$, if $t_{D_i} > (T-1) \times t_A$ and $A = 1$, then shortest-t_D-first (StF) schedule is optimal in terms of idle time for the second machine.*

However, StF schedule does not necessarily produce the optimal schedule. Apart from idle times of machines, the length of the *head* and *tail* of the schedule determine the optimality of a schedule. For the problem under consideration, the length of the head is independent of the scheduling algorithm. The StF schedule puts the job with largest second machine time last. This results in bigger tail, producing a suboptimal makespan. Schedules generated by longest-t_D-first (LtF) algorithm on the other hand can produce idle time on the second machine but are successful in reducing the length of the tail. This is because the LtF algorithm puts smallest jobs at the end of the schedule. [14] shows that when number of jobs is less than or equal to number of instances of the second machine LtF produces optimal makespan. But LtF is not necessarily optimal when number of job exceeds number of instances of the second machine.

Proposition 4. [14] *$\forall i$, if $t_{D_i} > (T-1) \times t_A$ and $A = 1$ and $n > T$, then longest-t_D-first (LtF) schedule can be suboptimal.*

We propose a new heuristic that combines properties of StF and LtF:

1. Sort the jobs using StF strategy.
2. Pick the last T (number of instances of the second machine) jobs and reverse their order. If there are n jobs, the last T jobs are numbered $n-T, n-T+1, \ldots, n-2, n-1$ at the end of previous step. And $t_{D_{n-T}} \leq t_{D_{n-T+1}} \leq \ldots \leq t_{D_{n-2}} \leq t_{D_{n-1}}$. We reverse their order so that $t_{D_{n-T}} \geq t_{D_{n-T+1}} \geq \ldots \geq t_{D_{n-2}} \geq t_{D_{n-1}}$
3. Repeat the above step for jobs $n-2T, n-2T+1, \ldots, n-T-2, n-T-1$. Keep repeating step 3 moving towards the start of the schedule.

Below is the illustration explaining working of our heuristic algorithm:

Jobs	a	b	c	d	e	f	g	h	i	j	k	l	m	n
t_D	6	10	7	13	8	5	4	1	9	2	0	12	11	3
Applying StF (Step 1)	k	h	j	n	g	f	a	c	e	i	b	m	l	d
Reversing order of the last 4 jobs (Step 2)	k	h	j	n	g	f	a	c	e	i	d	l	m	b
Repeated application of step 3	k	h	j	n	g	f	i	e	c	a	d	l	m	b
	k	h	f	g	j	n	i	e	c	a	d	l	m	b
	h	k	f	g	j	n	i	e	c	a	d	l	m	b

The workload consists of 14 jobs (a, b, ..., n). The configuration of the flowshop is $A = 1, T = 4$. The jobs are first sorted using StF algorithm. Then the order of the last 4 jobs is reversed. The algorithm then works its way towards the start of the schedule, reversing orders of 4 consecutive jobs. In the final step, only two jobs remain, jobs h and k. Their order is reversed too.

6 Performance Evaluation

So far we assumed that the values of t_A and t_{D_i} for each job (tape to be read) are known. In general, its hard if not impossible to calculate t_{D_i} accurately given the set of blocks on the tape that are to be read, since it requires accurate modeling of the tape drive(s). On the other hand, tape drive manufacturers do provide peak/average search (seek) rate and peak/average read rate. These values can be used to estimate t_{D_i}. The estimated value of t_{D_i} is denoted by $t_{D_i}^{estimated}$. Ideally, if $t_{D_j} < t_{D_k}$ then $t_{D_j}^{estimated} < t_{D_k}^{estimated}$ should hold. We evaluated three different schemes to compute $t_{D_i}^{estimated}$:

1. *Maximum Offset Estimate* For each tape find the offset of the farthest block to be read inside that tape. For each tape i, $t_{D_i}^{estimated}$ = maximum offset. This value is approximately proportional to the time it will take to rewind this tape under the tape drive model we use.
2. *Data Volume Estimate* For each tape i, $t_{D_i}^{estimated}$ = number of blocks read from the tape. This value is approximately proportional to the time it will take to read the blocks from this tape.
3. *Full Estimate* This estimation method combines the above two estimation methods. $t_{D_i}^{estimated}$ = seek rate $\times$ maximum offset + read rate $\times$ blocks read + seek rate $\times$ (maximum offset − blocks read).

Our experiments revealed that *data volume estimates* and *full estimates* help scheduling algorithms perform better than using *maximum offset estimates*. We also found that scheduling algorithms perform equally well whether *data volume estimates* are used or *full estimates* are used. This is because read times dominate seek times for the workloads we considered. We use *data volume estimates* for all scheduling algorithms since it has lower computing requirements.

We use a tape library simulator to execute the schedules created by various scheduling algorithm. Most of the literature [3, 17, 21, 22] uses a linear approximation of the locate time for tape drives. [7] found that such linear approximation is inaccurate. We use the analytical models of Exabyte's EXB-8505XL tape drive and EXB-210 tape library described in [6] in our tape library simulator. We use the *SORT* algorithm described in [9] for I/O scheduling when fetching data from the same tape.

Random Workload Fig. 1 shows the performance of various scheduling algorithms over a set of randomly generated workloads. The set contains 1000

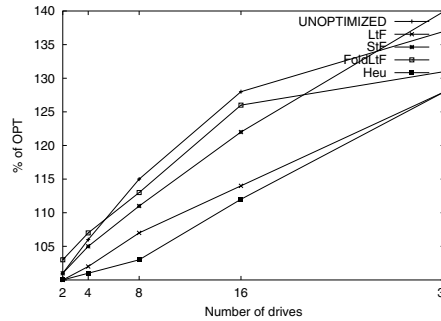


Fig. 1. Performance of various scheduling algorithms for random workloads.

workloads[1]. For each workload, we determine how many blocks to read from each tape by generating a random number[2] between 0 and the total number of blocks on the tape. Then for each tape, we generate N distinct block numbers randomly, where N is the number of blocks to be read from this tape.

The UNOPTIMIZED algorithm loads the tapes in random order. The FoldLtF algorithm is a heuristic proposed in [16] for job scheduling in a limited floor-space flow-shop environment. The algorithm first generates a list using LtF algorithm and then schedules jobs from both the ends of the list. The figure plots the makespan of each scheduling algorithm as percent of an OPT value. The OPT value is a lower bound on the makespan of the optimal schedule. The OPT value is then sum of the times to access each tape divided by the total number of drives in the system. Note that we use the same set of workloads for all the data points in the figure. Hence the OPT value is inversely proportional to the number of drives in the system. We find that performance of the scheduling algorithms is not within a constant factor of OPT (for the expected range of value of number of drives in the system), its a function of the number of drives in the system. Since LtF always outperforms StF, we conclude that the length of the tail of the schedule is more important than amount of idle time in the schedule for reducing makespan, The FoldLtF algorithm performs only slightly better than the UNOPTIMIZED case that too when number of drives in the system is comparatively higher. Our heuristic based algorithm always performs well due to a careful balance between idle times and length of tail achieved by our algorithm. Performance of LtF algorithm approaches performance of our algorithm when number of drives in the system is very low or very high. The reason LtF (and the UNOPTIMIZED case too) perform on par with our algorithm when number of drives is small is because there is very little scope for optimization in that case due to limited choice available to scheduling algorithms when fewer tape

[1] We found that varying number of instances of the workload does not change the results qualitatively.

[2] We use an inversive congruential generator for generating random numbers.

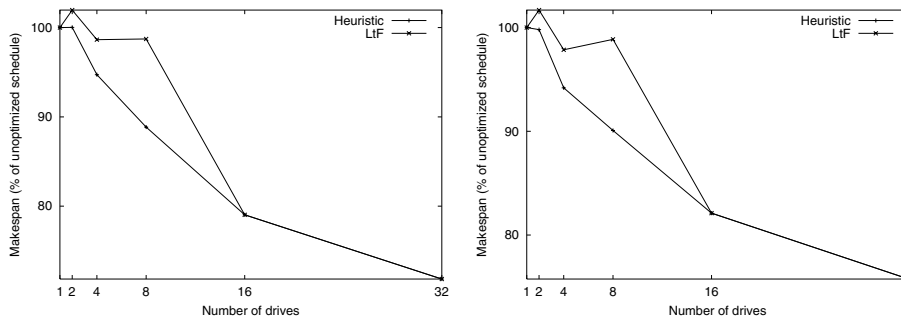
drives are used. As we saw earlier, LtF is optimal for large jobs when number of instances of the second machine are equal to or greater than number of jobs. When the number of drives is high, number of tapes to be loaded is close to/equal to number of drives available, making LtF optimal. But our algorithm clearly outperforms LtF when number of drives in the system is moderate (between 4 and 16). Note that this range of values of number of drives, is commonly found in a typical data management system handling large amounts of data. Since the performance of UNOPTIMIZED, StF and FoldLtF is considerably worse than LtF and our heuristic algorithm, we did not consider these algorithms for further performance studies.

Experimental Verification Using Sequoia 2000 Storage Benchmark We use the national dataset from the benchmark over a period of four years (200 weeks) and is about 64GB in size. The schema for the tables used in these queries is:

```
create RASTER(location=box, time=int4, band=int4, data=int2[][]);
```

time is a four byte integer and denotes the half-month over which the raster image was captured. The *location* attribute is the bounding box for the raster data. *band* is the wavelength band at which the data was captured. *data* is a two dimensional array of size $10240X6400$ of two byte integers at a spatial resolution of $0.5kmX0.5km$. All the raster images are stored chronologically sorted, since they were captured such. Raster images for a half-month are not sorted in any particular order.

A query type represents an access pattern on the dataset. A query is an instance of a query type. In general multiple access patterns are observed on a typical dataset. Access patterns are executed with different frequencies [2, 11]. In order to capture this phenomenon, we first define variety of access patterns (query types) on the dataset. Then we create different query mixes using these query types by manipulating number of different queries for each query type in the mix. We use two types of queries. *Query Type 1* selects all images belonging to a band. The data of interest is spread over the entire set of tapes that store the dataset. *Query Type 2* selects all images belonging to a half-month. The data of interest is localized in a few tapes of the set of tapes that store the dataset. A query mix is generated using two parameters: *Number of queries* denotes the total number of queries that this mix will consist of. *Query type percentages* represent the percentage of query instances that belong to each query type. The number of queries determines the accuracy of the query mix generation process. For all query types, if the number of distinct query instances that belong to a query type is n and the query type percentage is p, the mix should contain at least $\frac{n}{p}$ queries. This assures that the expected number of occurrences for any query instance for a query type is at least 1. We evaluate two different query mixes: *Query mix 1* consists of majority (90%) of queries from query type 1. *Query mix 2* has equal mix of queries from query type 1 and query type 2.



(a) Query mix 1 (b) Query mix 2

Fig. 2. Experimental results for Sequoia 2000 storage benchmark

Fig. 2 shows the performance results of our heuristic algorithm against LtF algorithm. The time taken to execute a query mix by an algorithm is plotted as a percent of the time taken by a naive scheme that does not do any scheduling. The results show that our algorithm performs consistently well. Note that for 16 and 32 drives case, the number of tapes from which data is read for a query is less than the number of drives in the system. Since LtF has already been proved to be optimal in that case, LtF performs equally well when compared to our algorithm. When number of drives in the system is moderate, our algorithm clearly outperforms LtF. The gains in performance are due to a balanced optimization of both drive idle times and the size of the tail of the schedule.

7 Conclusions

This paper investigated issues in optimizing I/O time for a query whose data resides on automated tertiary storage containing multiple storage devices. We modeled the problem as a limited storage parallel two-machine flow-shop scheduling problem with additional constraints. The paper presented analytical results that provide insight to the problem. We presented a heuristic algorithm for scheduling data from a tape library. Our performance results show impressive gains for synthetic as well as real workloads.

References

[1] Carey, M. J., Haas, L. M., and Livny, M. Tapes hold data, too: Challenges of tuples on tertiary storage. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., 1993), ACM Press, pp. 413–417.

[2] Chen, L. T., Drach, R., Keating, M., Louis, S., Rotem, D., and Shoshani, A. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. *Information Systems 20*, 2 (April 1995), 155–183.

[3] Drapeau, A. L., and Katz, R. H. Striping in large tape libraries. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference* (San Diego, CA, 1993), IEEE Computer Society Press.

[4] Fox, S., Prasad, N., and Szezur, M. NASA's EOSDIS: an integrated system for processing, archiving, and disseminating high-volume earth science imagery and associated products, July 1996.

[5] Golubchik, L., Muntz, R. R., and Watson, R. W. Analysis of striping techniques in robotic storage libraries. In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems* (Monterey, CA, 1995), IEEE Computer Society Press, pp. 225–238.

[6] Hillyer, B. K., Rastogi, R., and Silberschatz, A. Scheduling and data replication to improve tape jukebox performance. In *Proceedings of the 15th International Conference on Data Engineering* (Sydney, Australia, 1999), IEEE Computer Society Press, pp. 532–541.

[7] Hillyer, B. K., and Silberschatz, A. On the modeling and performance characteristics of a serpentine tape drive. In *Proceedings of 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Philadelphia, Pennsylvania, 1996), ACM Press, pp. 170–179.

[8] Hillyer, B. K., and Silberschatz, A. Random I/O scheduling in online tertiary storage systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Canada, 1996), ACM Press, pp. 195–204.

[9] Hillyer, B. K., and Silberschatz, A. Scheduling non-contiguous tape retrievals. In *Proceedings of Sixth NASA Goddard Conference on Mass Storage Systems and Technologies and Fifteenth IEEE Mass Storage Systems Symposium* (University of Maryland, College Park, MD, March, 1998), IEEE Computer Society Press.

[10] Inmon, B. The Role of Nearline Storage in the Data Warehouse: Extending your Growing Warehouse to Infinity. Technical white paper, 1999. Provided by StorageTek. http://billinmon.com/library/whiteprs/st_nls.pdf.

[11] Jagadish, H. V., Lakshmanan, L. V. S., and Srivastava, D. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. In *Proceedings ACM SIGMOD International Conference on Management of Data* (Philadephia, Pennsylvania, 1999), ACM Press, pp. 37–48.

[12] Johnson, S. M. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly 1*, 1 (March 1954), 61–68.

[13] Kobler, B., Berbert, J., Caulk, P., and Hariharan, P. C. Architecture and design of storage and data management for the nasa earth observing system data and information system (eosdis). In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems* (Monterey, CA, 1995), IEEE Computer Society Press, pp. 65–76.

[14] More, S., and Choudhary, A. Scheduling Queries on Tape-resident Data. Tech. Rep. CPDC-TR-2000-01-001, Center for Parallel and Distributed Computing, Northwestern University, January 2000. http://www.ece.nwu.edu/cpdc/TechReport/1999/11/CPDC-TR-2000-01-001.html.

[15] More, S., and Choudhary, A. Tertiary storage organization for large multidimensional datasets. In *8th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies and 17th IEEE Symposium on Mass Storage Systems* (College Park, MD, March 2000), IEEE Computer Society Press, pp. 203–209.

[16] More, S., Muthukrishnan, S., and Shriver, E. Efficiently sequencing tape-resident jobs. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, June 1999), ACM Press, pp. 33–43.

[17] Myllymaki, J., and Livny, M. Disk-tape joins: synchronizing disk and tape accesses. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada, 1995), ACM Press, pp. 279–290.

[18] APB-1 OLAP Benchmark, Release II, November 1998. OLAP Council.

[19] Pinedo, M. *Scheduling Theory, Algorithms and Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[20] Sarawagi, S. Database systems for efficient access to tertiary memory. In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems* (Monterey, CA, 1995), IEEE Computer Society Press, pp. 120–126.

[21] Sarawagi, S. Query processing in tertiary memory databases. In *Proceedings of 21th International Conference on Very Large Data Bases* (Zurich, Switzerland, 1995), Morgan Kaufmann, pp. 585–596.

[22] Sarawagi, S., and Stonebraker, M. Reordering query execution in tertiary memory databases. In *Proceedings of 22th International Conference on Very Large Data Bases* (Mumbai (Bombay), India, 1996), Morgan Kaufmann, pp. 156–167.

[23] Stonebraker, M. Managing persistent objects in a multi-level storage. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (Denver, Colorado., 1991), ACM Press, pp. 2–11.

Logging RAID – An Approach to Fast, Reliable, and Low-Cost Disk Arrays

Ying Chen, Windsor W. Hsu, and Honesty C. Young

IBM Almaden Research Center
{ying, windsor, young}@almaden.ibm.com

Abstract. Parity-based disk arrays provide high reliability and high performance for read and large write accesses at low storage cost. However, small writes are notoriously slow due to the well-known *read-modify-write* problem. This paper presents logging RAID, a disk array architecture that adopts data logging techniques to overcome the small-write problem in parity-based disk arrays. Logging RAID achieves high performance for a wide variety of I/O access patterns with very small disk space overhead. We show this through trace-driven simulations.

1 Introduction

Several key characteristics of a redundant array of independent disks (RAID), e.g., high I/O bandwidth, high system reliability, and low cost, have made them attractive to many different types of applications. Traditionally, RAID systems are classified into a number of RAID levels, each corresponding to a different arrangement of data and parity on disk. The common RAID levels include RAID level 0, 1, 3, 4, and 5 ("RAID-x" for short). [1] gives a detailed description of different RAID levels. We describe these RAID levels briefly here.

RAID-0 stripes data across a set of disk drives in a round-robin fashion. The unit of striping is typically a multiple of disk sector size. This unit is commonly referred to as "stripe unit". Each row of stripe units across all the drives in the array is called a "stripe". RAID-0 allows for parallel disk accesses, but it is unreliable. RAID-1 deals with the reliability issue by duplicating each data block on two separate drives, i.e., "data mirroring". RAID-1 tolerates any single drive failure, but it also doubles the storage capacity cost.

RAID-3, 4, and 5 are parity-based RAID architectures, where redundant data information is stored in the form of *parity*, which is computed as an XOR of all the data it protects. Parity-based RAIDs provide single-drive failure protection at low storage capacity cost[1]. They perform well for workloads with a large number of reads and/or large, stripe-aligned writes, i.e., writes that completely overwrite one or multiple RAID stripes, but they suffer the well-known *read-modify-write* problem for small writes. This is because for each small write, the parity-based RAID must read the old data block and the old parity information,

[1] Only $\frac{1}{D}$th of the total storage is needed to store parity information. Here D is the number of disks in a RAID system excluding parity disk.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1302–1311, 2000.
© Springer-Verlag Berlin Heidelberg 2000

compute the new parity, and write the new data and the parity blocks. Four disk I/O operations per small write is common in parity-base RAID systems. The problem does not exist for large, stripe-aligned writes, since the entire stripe will be overwritten, there is no need to read the old data and parity information. RAID-3, 4, and 5 have relatively small differences as highlighted in [1]. We do not discuss them due to space limitations. In this paper, we use RAID-5 to represent parity-based RAIDs. Our techniques are applicable to RAID-4 as well.

The logging RAID architecture is designed to overcome the RAID-5 small-write problem while guaranteeing high performance for many other read and write access patterns. The cornerstone of this architecture is the data logging technique that has been used in several different types of systems [2,3,4]. Logging RAID bundles small writes into large RAID-5 stripes using a small Non-Volatile-Memory (NVM) buffer. We flush the NVM buffer to the end of a small temporary log on disk based on a predetermined threshold. Writes to the log are always RAID-5 stripe-aligned, and hence efficient. Data blocks stored in the log are moved to their permanent disk locations later on during system idle time to ensure subsequent efficient read accesses.

In the remainder of the paper, we begin with a description of the logging RAID scheme in Section 2. Section 3 presents a set of trace-driven simulation results. Due to space limitations, we do not discuss related work in this paper. Interested readers can find a relatively detailed description of the related work in [5]. Finally, we draw conclusions in Section 4.

2 The Logging RAID Architecture

To improve write performance, a number of systems have chosen to write data in the form of a log. However, if care is not taken, data logging can slow down reads, since the layout suitable for writes may not be appropriate for reads. For instance, the workloads generated by data-mining and decision support systems [6] often contain a sweep of random writes followed by large sequential reads/scans. If the randomly updated data blocks are grouped together on disk, sequential scans may perform poorly.

Logging RAID exploits the data logging idea to optimize writes, while working hard to optimize reads. To optimize writes, logging RAID always writes data in large, RAID-5 stripe-aligned fashion. To optimize reads, an off-line data destaging and reorganization process is performed when the system is idle. In the rest of the discussion, we assume that the smallest unit addressable by the host system is one disk block or sector (512 bytes).

2.1 The Logging RAID Storage Layout

Conceptually, the disk space in logging RAID consists of a permanent storage area ("PA" for short) and a temporary log area ("LA" or "log" for short). The host system only sees the PA. Reads and writes are performed as if the LA does not exist. The LA is a small fraction of the total disk space in the disk array.

Both the PA and the LA are organized as RAID-5 and may reside on the same or different sets of disks in a given array. They can also use different stripe units and different numbers and types of disks.

The log serves as a "fast write cache" in logging RAID. Writes to the log are always appended to the end. For in-line small writes, logging RAID first accumulates them in a small NVM buffer, which contains one or more RAID-5 stripes in the log. Each RAID-5 stripe in the NVM buffer is called "a stripe buffer". One or more stripe buffers are flushed to the log whenever the NVM buffer is full. So the unit of flushing is always one or more stripe buffers. This guarantees that writes to the LA are always RAID-5 stripe-aligned, hence fast. Normal writes can return as soon as the data is written to the NVM buffer, so the I/O response time is short. To simplify our discussion, we assume that the NVM buffer contains one RAID-5 stripe in the rest of the paper.

Storing data in this log-structured layout is good for writes. However, as discussed earlier, some read patterns (e.g. seqeutial reads) may suffer poor performance. To avoid subsequent poor sequential read performance, logging RAID moves data stored in the LA to the PA during system idle time. Note that the underlying assumption is that the data layout in the PA is efficient for sequential reads. We believe that this is reasonable since the data layout in the PA is typically determined by host file system or applications. Most of today's file systems and applications such as databases employ a fair amount of optimizations on data placement, such as allocating contiguous disk blocks for data in the same file. Currently, we do not exploit other reorganization optimizations, such as optimizing data layout based on runtime request access patterns [7].

2.2 The Mapping Structures

In logging RAID, a disk block may reside in the LA or the PA. The data blocks in the LA are always the most up-to-date, since the new updates are always written to the LA first (with one exception as discussed in Section 2.3.). Logging RAID uses a log map, implemented using a hash table, as an internal mapping structure to keep track of the blocks that are in the LA. On receiving a read request for block x, logging RAID first checks if there is a hash entry corresponding to x. If so, x is read from the disk block address indicated by the hash entry. Otherwise, x must be in the PA and its location is x. Writes are somewhat tricky since a disk block can be moved from PA to LA or from one location to another location within LA. Section 2.3 explains in detail how writes are handled.

Note that the typical request size between the host system and the disk subsystem is often not as small as one disk sector. In fact, most file systems format disk space using a block size of several KBs. This suggests that the unit of relocation between the PA and the LA or within the LA itself can be much larger than a disk sector. Using large relocation units also reduces the mapping size, since logging RAID only needs to maintain one hash entry per relocation unit rather than per disk sector. The hash table size is $c \times \frac{la}{r}$, where c is the number of bytes per hash table entry, la is the log size, and r is the relocation

unit size. For example, with a 2 GB log space and 2 KB relocation unit size, the total hash table size is around 16 MB (assuming $c = 16B$).

If the host system only updates a subset of the disk blocks in a relocation unit, logging RAID will first read the entire relocation unit from either the LA or the PA (depending on where the disk blocks are) into the NVM buffer before the update can take place. Clearly, using a small relocation unit will reduce disk bandwidth usage but may significantly increase the memory requirement for the hash table. Logging RAID must choose an appropriate relocation unit to balance such tradeoffs. We evaluate the effect of relocation unit size in section 3.

Logging RAID keeps the hash table in memory at all times. The hash table is also written to a designated disk location in the disk array during system idle time and normal system shutdown. To ensure that logging RAID can reconstruct the in-memory hash table after a system crash, the hash table itself is also logged and written to disk in a similar fashion to data block updates. In other words, logging RAID uses an NVM buffer to accumulate hash table changes and a dedicated log space to store the hash table on disk. Whenever there is an update to the hash table, the update is first written to the NVM buffer. When the buffer is full, it is flushed to the end of the hash table log on disk. This NVM buffer is separate from the NVM buffer used to accumulate the RAID-5 data writes. Logging RAID also periodically writes the entire in-memory hash table to disk (called "checkpointing" in logging RAID). Checkpoints are infrequent, so they rarely cause visible performance impact. After the hash table is checkpointed on disk, logging RAID can recycle the log disk space for the hash table before that checkpoint. Between two consecutive checkpoints, only changes to the hash table need to be written. During system recovery, logging RAID retrieves the mapping information from the most recent checkpoint, then uses the hash table log and NVM to perform a roll-forward to reconstruct the system states.

2.3 Logging RAID Operations

In-line writes Upon receiving a small write to a set of disk blocks, logging RAID first checks if the relocation units that the disk blocks reside in are in the NVM. If so, logging RAID updates the corresponding relocation units directly in the NVM. Otherwise, logging RAID consults the hash table and reads the disk relocation units either from the LA or the PA into the NVM (this read is needed only if the updates are not full relocation unit updates) and updates them. The writes can return to the host once they are written to the NVM. One exception to the scheme above is that for large, RAID-5 stripe-aligned writes, instead of relocating the data blocks in the stripe from the PA to the LA or within the LA, logging RAID writes the stripe directly to the PA. This avoids moving data around later without sacrificing write performance. For each write, if the disk blocks are written to a new location, the hash table is updated accordingly.

In-line reads Upon receiving a host read request, logging RAID first checks if the data is in the NVM. If so, the read can be satisfied from the NVM buffer. Otherwise, logging RAID checks the hash table to see if the data has been

relocated to LA. If so, logging RAID fetches the data from the LA locations indicated by the hash entry. Otherwise, the requested data is read from the PA.
Background operation: Idle time data destaging Logging RAID destages the relocation units from LA to PA whenever there is a detected system idle time. Logging RAID destages one stripe at a time until the amount of free space reaches a predetermined threshold. Currently, logging RAID selects the destaging RAID-5 stripe from the head of the log. Since new writes are always appended to the end of the log, the head of the log contains the "oldest" data. [5] discusses several more sophisticated schemes. We skip them here due to space limitations. System idle time detection is also an interesting problem [8]. Currently, logging RAID starts the idle time activities whenever the system is idle for D seconds.
In-line data destaging In-line data destaging may be necessary when the log becomes full during NVM flushing. Logging RAID chooses to perform one of the following two actions based on the system state at the time of NVM flushing: 1. write new data directly to PA as in a RAID-5 system. 2. queue the new writes and move some RAID-5 stripes from the LA to the PA to make room for the new writes. After moving some RAID-5 stripes from the LA to the PA, logging RAID writes the NVM buffer to the LA and the queued requests can proceed.

In theory, if moving data from the LA to the PA is fast, e.g., when the RAID-5 stripe to be moved contains a lot of "holes" (the holes are created due to multiple updates on the same stripe), queuing the new requests could be more benefitial than direct updates to the PA. On the contrary, if the selected RAID-5 stripe is mostly full, moving it back to the PA could generate a lot of read-modify-write operations. So writing new data directly to the PA could be a win. Currently, logging RAID chooses to queue writes and perform in-line data destaging only if the selected RAID-5 stripe from the LA has at least half of the stripe empty. Otherwise, logging RAID writes the new data to the PA directly.

3 A Trace-Driven Simulation Study

To evaluate the effectiveness of logging RAID, we carried out a trace-driven simulation study. We built a disk array simulator based on DiskSim [9], a configurable disk system simulator. In our evaluation, we studied the behavior of logging RAID under different situations and compared the performance of logging RAID with RAID-1 and RAID-5. Due to space limitations, we do not present all the results in this paper. More detailed results can be found in [5].

3.1 Experimental Setups and Traces

We used several traces from different systems to drive our simulations. **CELLO** is a three-month trace gathered on a time-sharing HP-UX system in a real world engineering environment. **SNAKE** is a two-month trace collected on a HP-UX cluster file server. In addition, we have two TPC-C [10][2] traces gathered on

[2] Because our TPC-C benchmark setup has not been audited per the benchmark specification, our workload is technically not TPC-C benchmark workload and should

a multiprocessor PC server running DB2. TPC-C benchmark is an industry standard benchmark used to measure the database system performance for on-line transaction processing (OLTP) type of workloads. The difference between the two TPC-C traces is on the database buffer pool size configurations in DB2. One trace used a 32 MB buffer pool, and the other used 128 MB. Different DB2 buffer pool configurations can result in different request sequences at the disk array controller due to DB2 caching effects. We are interested in how logging RAID performs with these different TPC-C request streams.

Due to long simulation times and numerous combinations of simulations tests, we only used a fraction of the Cello and TPC-C traces in our simulations. Specifically, we used the first one-third of the TPC-C traces and the first half of the Cello trace. Table 1 summarizes the key characteristics of our traces. We especially focused on the write characteristics of these traces. In both the Cello and Snake traces, almost all the writes are less than or equal to 8 KB. In the TPC-C traces, the request size is always 4 KB.

Trace characteristics	**Snake**	**Cello**	**TPCC-32**	**TPCC-128**
# of requests (in millions)	12.0	25.4	7.2	5.5
# of reads (in millions)	5.3	11.0	4.7	3.6
# of write (in millions)	6.7	14.4	2.5	1.9
Ave read request size	6 KB	6 KB	4 KB	4 KB
Ave write request size	7 KB	6.5 KB	4 KB	4 KB
Write footprint	530742	2022709	1862556	1828328

Table 1. The key characteristics of the traces. Here, TPCC-X is the TPC-C trace with X MB buffer pool size. The write footprint is the number of distinct 1 KB disk blocks (the smallest write unit in all the traces).

To study the logging RAID behavior, we varied the individual logging RAID tuning parameters one at a time, i.e., the relocation unit sizes and the log space. We also varied the RAID-5 stripe unit size. We found that logging RAID is insensitive to the stripe unit size as long as the unit is larger than 8 KB. In the results shown in this paper, we used 16 KB as the RAID-5 stripe unit. We used smaller number of disks for Snake since it has a smaller footprint. Table 2 lists the parameters used in our sensitivity study. The bold items are the baseline settings for the results shown in this paper. We focus on the relative performance of logging RAID under different situations rather than the absolute performance. Our performance metrics is the average I/O response time.

only be referred to as TPC-C-like. In the rest of this paper, when TPC-C is used to refer to our workload, it should be taken to mean TPC-C-like. Details about the TPC-C traces and the collection methodology can be found in [11].

Simulation parameter	Settings
Relocation unit size (KB)	**2**, 4, 8, 16, 32
Fraction of disk space for log	1%, **5%**, 10%, 20%, 40%
NVM size (KB)	64, 128, **256**, **512**, 1024, 2048, 4096
Stripe unit (KB)	4, 8, **16**, 32, 64
Number of disks	8 for Snake, 16 for others
Disk model	HP C2249A
Request scheduling policy	**FCFS**, SSTF, ELEVATOR, VSCAN
Disk RPM	**5400,** 7200, 10000
SCSI bus transfer rate	**10 MB/s**, 20 MB/s, 40 MB/s

Table 2. The parameter settings used in the sensitivity study. The bold items are the settings used in the results shown in this paper.

3.2 Simulation Results

Sensitivity to Relocation Unit Sizes Figure 1 shows the performance of logging RAID with different relocation unit sizes for Snake and Cello with a 256 KB NVM stripe buffer. A general trend in these results is that relatively small relocation units give better performance. This is because almost all writes are less than or equal to 8 KB in both of these traces. In logging RAID, if a write only updates a portion of a relocation unit, a pre-read of the entire relocation unit is required. So using relocation unit sizes larger than 8 KB may waste disk bandwidth due to pre-reads and writes of unchanged data. Since our TPC-C traces only contain 4 KB requests, we use 4 KB as the relocation unit size.

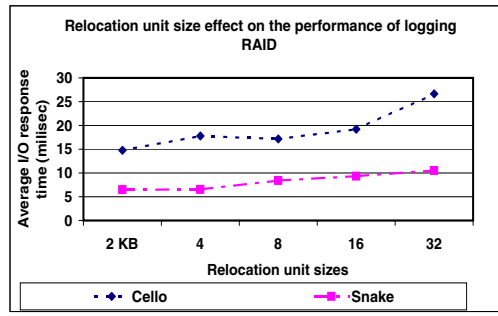


Fig. 1. Sensitivity of logging RAID performance to relocation unit sizes.

Sensitivity to Log Disk Space We varied the log space between 1% and 40% of the total disk space in the system. As discussed in Section 2, as long as the log is large enough to buffer the typical bursty writes, logging RAID will perform well. With a small log space, logging RAID has to destage some of the relocated units in-line, lengthening queuing delays for the host requests. Once the log space is

above 5%, the in-line destaging can be largely avoided as shown in Figure 2. It is clear that with a log space that is 1% of the total disk space, both Cello and Snake perform a little worse than the other cases. This is because some number of in-line destaging occured with this small log, causing host requests to wait whenever the in-line destaging occurs. For TPCC-32, 1% of the total disk space is sufficient to store the small writes. We did not repeat the tests with TPCC-128, since TPCC-128 contains fewer writes than TPCC-32.

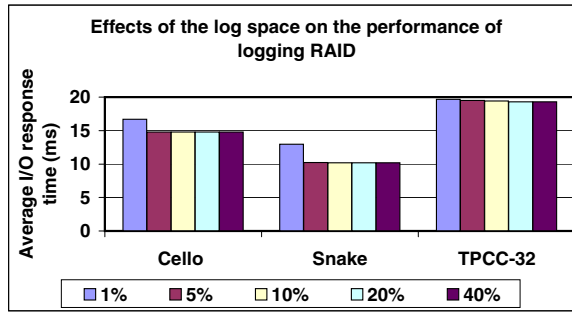


Fig. 2. Sensitivity of logging RAID performance to log space.

Note that shortage of log space can potentially affect the performance of logging RAID, but not to the point of being worse than a normal RAID-5 system. This is because in the worst case, logging RAID will simply write data directly to the PA, as does a normal RAID-5 system.

Comparison with RAID-1 and RAID-5 In this section, we compare the performance of logging RAID with RAID-1 and RAID-5 with a 512 KB NVM buffer size. For RAID-1 and RAID-5, we applied some optimizations on writes as detailed in [5]. We also selected the best stripe unit size and the number of disks for RAID-1 and RAID-5. We chose 64 KB stripe unit for RAID-1, and 4 KB and 8 KB for RAID-5 for TPC-C traces and HP traces respectively. In logging RAID, we used 5% of the total disk space for the log in the Cello and Snake tests, and 1% in the TPC-C tests.

Figure 3 compares the overall performance of different disk array systems, as well as break-downs by read and write performance. Overall, logging RAID gives the best performance in all the tests. It is encouraging to see that in many cases logging RAID performs much better than RAID-1. This is not a surprise since logging RAID can bundle multiple small writes into one large write, while this is not possible with RAID-1. When compared with RAID-5, logging RAID doubles (or even triples) the RAID-5 performance for both Snake and Cello.

Logging RAID did not improve TPC-C performance as much as it did the other two workloads. This is because our TPC-C traces are read-intensive, only 34% of the total requests are writes, while Cello and Snake are both write in-

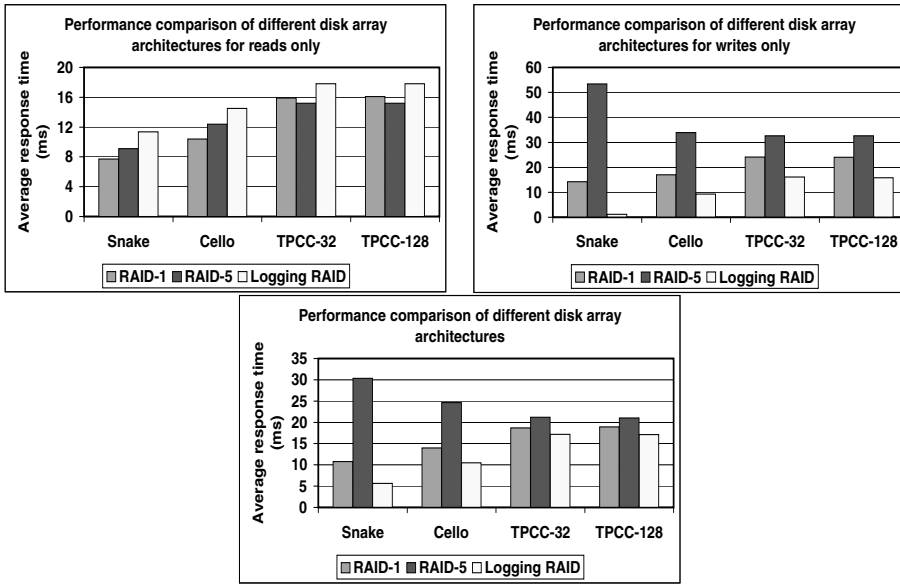


Fig. 3. Performance comparison of RAID-1, RAID-5 and logging RAID for read-only (top-left), write-only (top-right), and overall (bottom) situations.

tensive (more than 60% of requests are writes). Nevertheless, logging RAID is still more than 20% better than RAID-5 for TPC-C.

Since logging RAID relocates small data blocks between PA and LA, when reads (especially sequential reads) occur before the data are moved back from LA to PA, the read performance may suffer. To simulate such scenario, we examined the read performance alone in our gathered simulation results. We expect a degraded read performance in our results. Indeed, Figure 3 shows such effects through a breakdown of read performance and write performance. Clearly, logging RAID degraded read performance by a small percentage (from 10% to 24%) when compared to the RAID-5 read performance.

In an ideal case, if there is enough idle time for logging RAID to destage data from LA to PA before reads, both reads and writes can be done efficiently. When there is not enough idle time to move data from LA to PA, but the writes dominate the workloads, logging RAID also can perform very well. The problematic situation in logging RAID is when there is not enough idle time and yet reads dominate the workload. Moreover, many data blocks read have been randomly updated, and hence moved from the PA to the LA. In such cases, logging RAID could perform worse than a normal RAID-5.

There are several potential solutions to this problem. One is to adopt some dynamic access pattern detection mechanism to detect such workload patterns, and instruct logging RAID not to perform logging optimizations at all. Another possible solution is to select a larger relocation unit size. However, the relocation

unit size must strike a balance between wasting disk bandwidth on writes and maintaining data sequentiality for reads. Our relocation unit size sensitivity tests did not include the worst case workloads. Such tests may be done in the future.

4 Conclusion

Logging RAID employs the logging technique to provide superior performance under a wide range of workloads. It tolerates any single disk failure and incurs low storage hardware cost. The key techniques in logging RAID include: 1. Dynamic relocation of data blocks between a permanent storage area and a log area. 2. Batching small writes into large, sequential writes. Our trace-driven simulation results showed that logging RAID performs well with diverse workloads. The extra storage space required is small. Logging RAID outperforms all the disk array schemes that we simulated. Depending on the workload characteristics, logging RAID can more than double RAID-5 performance.

References

1. D. Patterson, G. Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.
2. M. Rosenblum and J. Ousterhout. The design and implementation of a Log-Structured file system. *ACM TOCS*, 10(1):26–52, Februray 1992.
3. J. Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM TOCS*, 14(1):108–136, February 1996.
4. J. Menon. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 167–178, August 1995.
5. Y. Chen, W. Hsu, and H. Young. Logging RAID – an approach to fast, reliable, and low-cost disk arrays. RJ 10161, IBM Almaden Research Center, October 1999.
6. Transaction Processing Performance Council. TPC benchmark D standard specification. TR, Waterside Associates, Fremont, CA.
7. J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of Log-Structured file systems with adaptive methods. In *Proceedings of the sixteenth ACM symposium on Operating system principles*, Saint Malo, France, 1997. ACM Press.
8. R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proceedings of Winter 1995 USENIX*, pages 201–222, New Orleans, LA, 1995.
9. G. Ganger, B. Worthington, and Y. Patt. The DiskSim simulation environment. Technical Report http://www.ece.cmu.edu/ ganger/disksim, University of Michigan, February 1998.
10. Transaction Processing Performance Council. TPC benchmark C standard specification. Technical report, Waterside Associates, Fremont, CA.
11. W. W. Hsu, A. J. Smith, and H. C. Young. Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks. TR CSD-99-1070, Computer Science Division, UC Berkeley, November 1999.

Topic 21
Problem Solving Environments

José C. Cunha, David W. Walker, Thierry Priol, and Wolfgang Gentzsch

Topic Chairmen

Introduction

This workshop encompasses several aspects of current research on Problem Solving Environments (PSE).

A PSE is an integrated computing environment for supporting the complete life cycle of design, development, and execution within a specific application domain. A PSE must assist the user in the design and evaluation of the more adequate solutions and problem-solving strategies. It must also support the development of rapid and efficient prototypes to ease the experimentation.

This idea of a PSE has been among us for several decades. There are already several fully developed PSEs in distinct areas, such as the automotive and aerospace industries, and PSEs are being developed in many research projects. Modern PSEs increasingly depend on an adequate integration of a diversity of heterogeneous components such as sequential, parallel or distributed problem solvers, tools for data processing, advanced visualization, computational steering, and access to large databases and scientific instruments.

Recently there has been an increasing awareness to this topic due to the need to exploit the enabling technologies that will allow the handling of more complex simulation models and larger volumes of generated data, higher degrees of human computer interaction, and more effective forms of cooperation among users in collaborative distributed environments.

The main goal of this workshop is to promote a discussion of the main issues involved in the design, implementation and application of PSEs, and to contribute towards future developments concerning PSEs.

The Papers in This Workshop

The papers in this workshop provide a global perspective of current work on PSE by addressing the following relevant issues:

- Coupling of multi-disciplinary simulation codes
- Integration of sequential and parallel programs
- The role of visualization systems in PSEs
- Interaction and computational steering
- Large scale Web-based environments
- Architectures of generic and domain independent PSEs

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1313–1314, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Overall, these five papers discuss approaches which are representative of the state-of-the-art and help identifying the trends in the development of future PSEs.

The first paper 'AMANDA - A Distributed System for Aircraft Design' by Kersken et al. discusses how a component-based framework was designed and how it is being used for the integration of coupled sequential and parallel programs. The paper identifies the main application requirements concerning the integration, the efficient massive data exchange between integrated programs, and the hierarchical structuring of a complex system. The paper describes the architecture of the TENT framework and the use of the MpCCI support library for code coupling. Experimental work on the development of two pilot applications in airplane and turbine design is described.

The second paper 'Problem Solving Environments: Extending the Role of Visualization Systems' by Wright et al. presents a discussion on how visualization systems can evolve concerning the aspects of collaborative working, data management and the usability of dataflow visualization environments. The role of Modular Visualization is emphasized as an approach to provide more generic and open architectures for PSEs. The above aspects are discussed and illustrated through examples using the IRIS Explorer system. An architecture is described for Web-based visualization using IRIS Explorer, HTML and VRML, aiming at high flexibility in visualization and user interaction.

The third paper 'An Architecture for Web-based Interaction and Steering of Adaptive Parallel/Distributed Applications' by Muralidhar et al. describes an ongoing effort towards the development of a Web-based collaborative environment supporting remote monitoring and control of adaptive scientific applications. The paper discusses the design of the distributed architecture of the environment and illustrates how its internal layers support the required interaction and steering capabilities.

The fourth paper 'Computational Steering in Problem Solving Environments' by Lancaster focusses on the design and implementation issues of computational steering in PSEs. The paper discusses the requirements posed by computational steering upon the architecture of a PSE. The paper describes experimental work on the implementation of a prototype PSE based on JavaBeans and CORBA and discusses the degree of steering achieved.

Finally, the paper 'Implementing Problem Solving Environments for Computational Science' by Rana et al. discusses the requirements to build a PSE that is easy to use, enables the development of new applications, or the uniform integration of existing application codes. The paper identifies the main functionalities of a PSE, concerning support for component composition and for resource management, and then it presents an overview of current efforts on PSEs. The authors propose a generic, domain independent, infrastructure for a PSE and then show how it is used to build two application dependent PSEs.

AMANDA - A Distributed System for Aircraft Design

Hans-Peter Kersken[1], Andreas Schreiber[1], Martin Strietzel[1], Michael Faden[1], Regine Ahrem[6], Peter Post[6], Klaus Wolf[6], Armin Beckert[4], Thomas Gerholt[2], Ralf Heinrich[5], and Edmund Kügeler[3]

[1] DLR, Simulation- and Softwaretechnology, 51170 Cologne, Germany,
[2] DLR, Institute of Fluid Dynamics, 37073 Göttingen, Germany,
[3] DLR, Institute of Propulsion Technology, 51170 Cologne, Germany,
[4] DLR, Institute of Aeroelasticity, 37073 Göttingen, Germany,
[5] DLR, Institute of Design Aerodynamics, 37073 Göttingen, Germany,
<first name>.<last name>@dlr.de
[6] GMD, Institute for Algorithms and Scientific Computing, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
<first name>.<last name>@gmd.de,

Abstract. In the AMANDA project a component-based framework for the integration of coupled technical applications, running distributed in a network, is developed. It is designed to deal with parallel and sequential programs and massive data exchange between the integrated programs. Two pilot applications will be implemented to show the feasibility of the chosen approach: a trimmed, freely flying, elastic airplane and an air-cooled turbine. Beside using the integration systems, the MpCCI[1] library is used for the coupling of the codes, each simulating single physical processes.

1 Introduction

An efficient design process for airplanes and propulsion systems requires the calculation of numerous variants for which, in the context of the product development, usually only a limited time window is available. A prerequisite for this task is the availability of high-quality procedures for the simulation of the physical processes involved, as well as their coupling, in order to be able to make predictions about physical and functional properties of the final product. In addition, a powerful soft- and hardware infrastructure must be available in order to be able to use these procedures fast and efficiently. Here a software integration system acts as an agent which manages the interaction of software tools. Otherwise the coupling of these tools would require time consuming human intervention. Additionally, it provides a uniform user interface to the integrated applications.

The AMANDA project has two main focuses: The first is to extent and enhance the software integration system TENT [9] which is a joint development of

[1] MpCCI is a trademark owned by GMD.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1315–1322, 2000.
© Springer-Verlag Berlin Heidelberg 2000

GMD and DLR. It was already used successfully in the project SUPEA [10] and is available as a prototype system [11]. The second focus is the implementation of pilot applications using this integration environment. Two applications have been selected:

- a trimmed, elastic, freely flying airplane and
- an air-cooled turbine.

The development of the integration framework is driven by requirements posed by these applications, e. g. the integration of parallel programs, the use of a hierarchical structure for setting up a process chain, or a control module to steer the execution of the workflow depending on internally calculated parameters.

The applications will be described in more detail in the next section, Sec. 3 gives an overview about the current state of the integration system, and Sec. 4 describes some new features of the integration system necessary for the implementation of the two AMANDA applications.

2 The AMANDA–Applications

2.1 Airplane Design

For the simulation of a trimmed, freely flying, elastic airplane the following programs have to be integrated into TENT:

- a CFD code, TAU [8] and FLOWer [4],
- a structural mechanics code (NASTRAN [7]) and a multi-body program (SIMPACK [5]), to calculate the deformation of the structure,
- a coupling module, to control the coupled fluid-structure computation,
- a grid deformation module, to calculate a new grid for the CFD simulation,
- a flight mechanics/controls module (build using an Object-Oriented modelling approach [6]), to set the aerodynamic control surface positions,
- visualization tools.

Figure 1 shows a possible scenario of the coupling. The process chain consisting of these codes is hierarchically structured. The lowest level contains the parallel CFD solver only. This CFD-subsystem consists of the flow solver itself and auxiliary programs to handle mesh adaptation. The next level, the aeroelastic-subsystem, comprises the CFD solver, the structural mechanics or multi-body code, and the grid deformation module. The data transfer from the CFD code to the structural mechanics code is performed via the MpCCI (former CoColib) [1] library which will be extended in order to deal with the interpolation used in fluid-structure interaction problems. For the solution of the coupled non-linear aeroelastic equations a staggered algorithm is implemented in the control process [2]. The highest level consists of the flight mechanics module coupled to the aeroelastic-subsystem. Each single code is additionally accompanied by its own visualization tool. The computation of a stable flight state typically proceeds as follows:

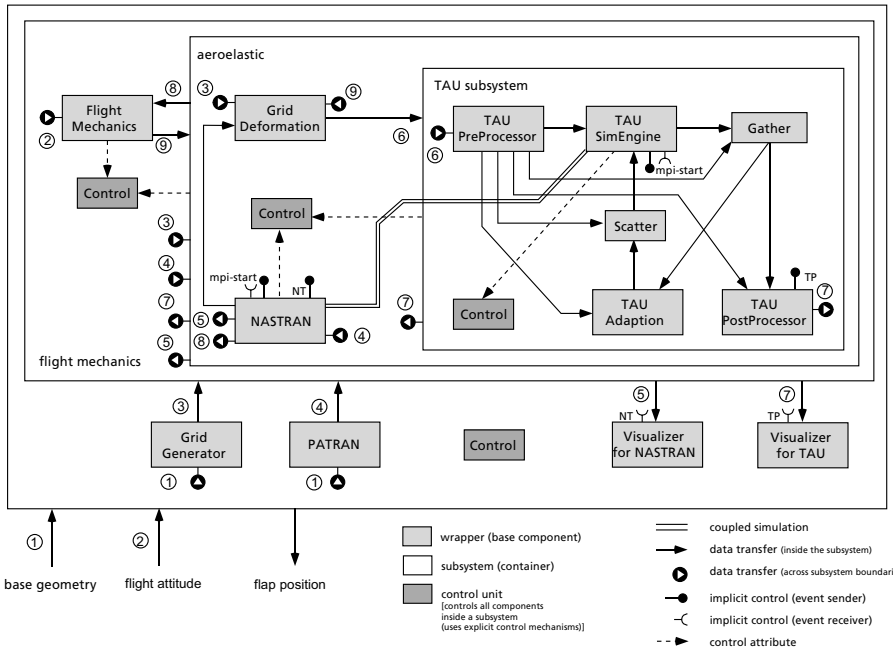


Fig. 1. Data flow for coupled CFD/structural mechanics/flight control system. The order of execution of the components is controled by the components labeled Control by means of attributes provided by the simulation components.

Starting by calculating the flow around the airplane, the pressure forces on the wings and the nacelle are derived. Theses forces are transfered to the structural mechanics code and interpolated to the structural mechanics grid using the MpCCI library. This code calculates the deformation of the structure which in turn influences the flow field around the airplane. At a final stage it is planned to extend the system and feed the calculated state into a flight mechanics/controls module to set the control surfaces accordingly and to obtain a stable flight position. This changes the geometry of the wings and requires therefore the recalculation of the flow field and the deformation.

As a prototype the coupled CFD/structural mechanics computation for a flexible wing has been realized without the integration system using file transfer for data exchange. The MPCCI library and the integration system will be used in near future.

2.2 Turbine Design

A new aspect in virtual turbine design is the simulation of flow inside the turbine in consideration of the heat load on the blades and the cooling. The numerical modeling of the position and size of cooling channels and holes in the blades are essential for the layout of an air-cooled turbine.

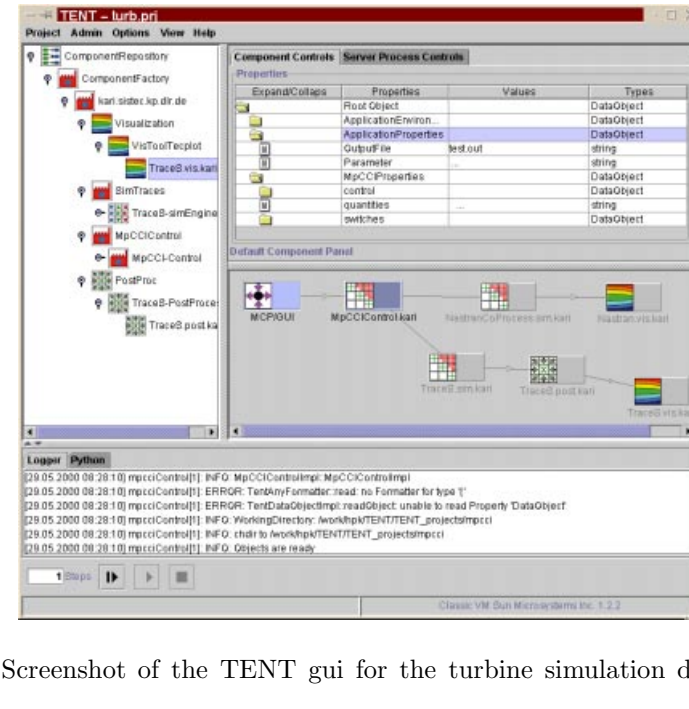


Fig. 2. Screenshot of the TENT gui for the turbine simulation described in Sec. 2.2

The CFD code TRACE [13], a 3D-Navier-Stokes-Solver for the simulation of steady and unsteady multistage turbomachinery applications, and a heat conduction problem solver (NASTRAN) are coupled to simulate the airflow through the turbine together with the temperature distribution inside the blades. For the coupling of the flow simulation and the heat conduction a stable coupling algorithm as been developed where TRACE delivers the temperatures of the air surrounding the blade and the heat transfer coefficients as boundary conditions to NASTRAN which in turn calculates the temperature distribution inside the blade. The temperature at the surface of the blade is returned to TRACE as boundary condition for the airflow.

A coupled simulation has been already realized using the MpCCI library for data exchange.

3 The Software Integration System

TENT is a framework for building and controlling complex technical workflows where the embedded application may be run on arbitrary machines in a network, i. e. the machines can be chosen depending on the application. It was developed in view of the integration of HPC applications and provides a graphical user interface to build process chains and control the integrated applications. Figure 2 show a screenshot for the simualtion scenario described in Sec. 2.2. The

TENT software system consists of four different packages as shown in Fig. 3. The software development kit (SDK) summarizes all interface definitions and libraries for the software development with TENT. The base system includes all basic services needed to run a system integrated with TENT. The facilities are a collection of high-level services, and the components consist of wrappers and special services for the integration of applications as TENT-components.

3.1 The Software Development Kit

TENT defines a component architecture and an application component interface on top of the CORBA object model. The TENT component architecture is inspired by the JavaBeans specification [3]. The data exchange interface as part of the software development kit allows the transparent data exchange between two TENT-components. It can invoke a data converter, which automatically converts data between the data formats supported by the integrated tools. For efficiency reasons CORBA is used only to coordinate the data transfer, i. e. only references to the data are exchanged using CORBA itself. These references are for example path names in a file system or port numbers and IP-addresses, if socket communication is employed. The transfer of the data themselves is performed in a more efficient way, depending on the situation at hand, e. g. by socket communication or file transfer.

3.2 TENT - Base System

The TENT base system consists of the components necessary to run the integrated system of applications. For portability reasons, these components are currently implemented in Java. The Master Control Process (MCP) is the main control instance in TENT. It realizes and controls system tasks such as process chain management, construction and destruction of components, or monitoring the state of the system.

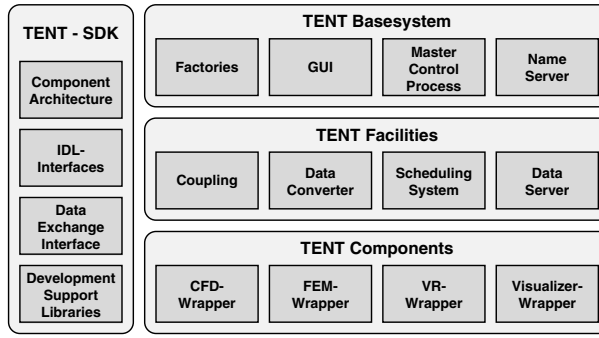


Fig. 3. Packages of the software integration system TENT.

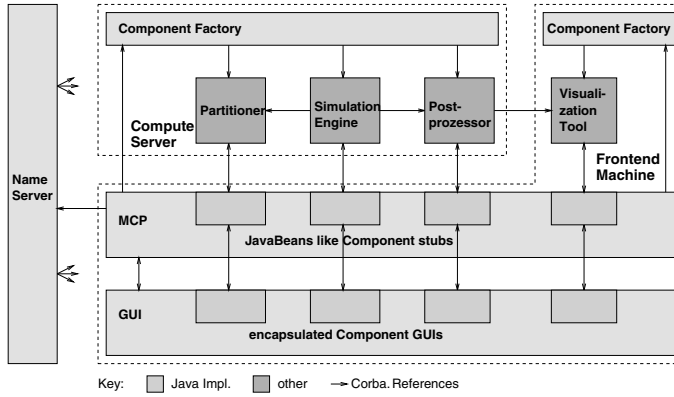


Fig. 4. Architectural overview of TENT.

The factories run on every machine in the TENT framework. They start, control, and terminate the applications on their machine. The naming service is an enhanced CORBA naming service. The relations between the base components are shown in Fig. 4.

3.3 TENT Facilities

In order to support higher-level workflows, the system offers more sophisticated facilities, e. g. data and job management(scheduling system) servers are planed. An important facility is the data converter which acts as a filter when integrated programs with incompatible data formats exchange information.

3.4 TENT Components

All applications must be encapsulated by wrappers to access their functionality in TENT. Therefore these wrappers have to be equipped with a CORBA interface to comply to the TENT component architecture. Depending on the level of access to the program, source code, API, or file I/O, the wrapper can be tightly coupled with the application, e. g. linked together with it, or must be a stand alone tool that starts the application via system services. Depending on the wrapping mechanism, the communication between the wrapper and the application is implemented by direct memory access, IPC mechanisms, file exchange, or any other suitable communication method

4 Impacts of the AMANDA–Applications on TENT

In this section new features which have been added to the integration system due to requirements from the AMANDA-project are discussed.

4.1 NASTRAN-Co-process

Every application not especially written for the TENT-systems has to be wrapped to be accessible by the system. For codes with source code access it is now a standard task. Some codes have already been integrated by changing the main routine only. The source code had to be modified to include control and communication commands. For NASTRAN just the opposite was true, neither the source code was accessible nor does an API exist. A NASTRAN run is controlled completely by an input file. Therefore, an additional program had to be written, a NASTRAN-co-process, to hook up NASTRAN to the integration system. The co-process is responsible for setting up NASTRAN input files, parsing NASTRAN output files to extract the results, and connecting to the integration system. Because this module was developed with its integration into TENT in mind, it was implemented as a library which methods can be called from a Python script. If NASTRAN is a part of an application coupled by the MpCCI the script can handle the control of the coupled application as well.

4.2 Strongly Coupled Multi-disciplinary Subsystems

In the context of AMANDA the coupling of multi-disciplinary simulation codes is a major issue. Therefore an interface to the coupling library MpCCI is provided. MpCCI coordinates the exchange of boundary informations between simulation applications working on adjacent grids. The communication between applications inside the integration system is usually controlled by CORBA mechanisms. In case of applications coupled by the MPI-based MpCCI, data are exchanged using the MPI-library without any interference of the integration system. The coupling algorithm itself, i. e. the number of time or iteration steps a code has to perform or the decision when to terminate the computation is not part of the MpCCI library. Hence, a control process is introduced into TENT to handle this task. A scripting language (Python) is used to simplify the development and the maintenance of this module. The flexibility offered by a scripting language is more important here than in other modules because setting up the coupling of two independent programs based on algorithms originally not intended for the use in coupled computation may require a lot of experimentation to find a good, i. e. converging and stable, coupling algorithm.

4.3 Hierarchical Structure

As can be easily seen in Fig. 1, without a structuring strategy it soon becomes impossible to handle the complexity of coupled problems. A design decision motivated by scenarios like this was to allow the encapsulation of existing workflows to become a single TENT-component which in turn may consist of other composite components. Each component has its own control process and by defining appropriate interfaces the complexity of a module is completely invisible to other components. The master control process is hence replaced by a set of independently working control processes in each module.

5 Conclusions

The TENT integration system allows to link software tools originally intended for stand alone use. By embedding them into the TENT framework, on the on hand, a uniform access to the tools is provided, on the other hand, data are transfered transparently between different tools without intervention from the user. The applications described show the flexibility and extensibility of the integration system. It will be used and extended in other, scientific and industrial, projects, e. g. virtual prototyping in the automotive industry [12], the simulation of a re-entry system, and combustion chamber modeling.

Acknowledgments

The work described in this paper is partially funded by the Hermann von Helmholtz-Gemeinschaft Deutscher Forschungszentren (HGF) and the german ministry for education and research under grant 01SF9822.

References

[1] R. Ahrem, P. Post, and K. Wolf. A communication library to couple simulation codes on distributed systems for multiphysics computations. In *Proceedings of ParCo99*, Delft, August 1999.
[2] A. Beckert. Ein Beitrag zur Strömung-Struktur-Kopplung für die Berechnung des aeroelastischen Gleichgewichtszustandes. ISRN DLR-FB 97-42, DLR, Göttingen, 1997.
[3] G. Hamilton. *JavaBeans API Specification*. Sun Microsystems Inc., July 1997.
[4] N. Kroll. The National CFD Project Megaflow - status report. In H. Körner and R. Hilbig, editors, *Notes on numerical fluid mechanics*, volume 60. Braunschweig, Wiesbaden, 1999.
[5] W. Krüger and W. Kortüm. Multibody simulation in the integrated design of semi-active landing gears. In *Proceedings of the AIAA Modeling and Simulation Technologies Conference*. Boston, 1998.
[6] D. Moormann, P. J. Mosterman, and G. Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, 3:115–126, April 1999.
[7] Nastran. http://www.macsch.com.
[8] D. Schwamborn, T. Gerholt, and R. Kessler. DLR TAU code. In *Procedings of the ODAS-Symposium*, June 1999.
[9] M. Strietzel, T. Breitfeld, T. Forkert, A. Schreiber, and K. Wolf. The distributed engineering framework TENT. to be published in: Vector and Parallel Processing – VECPAR2000. Lecture Notes in Computer Science, 2000.
[10] Supea. http://www.sistec.dlr.de/de/projects/supea.
[11] Tent. http://www.sistec.dlr.de/tent.
[12] C.-A. Thole, S. Kolibal, and K. Wolf. AUTOBENCH: Environment for the Development of Virtual Automotive Prototypes. In *Proceedings of 2nd CME-Congress (to appear)*, Bremen, September 1999.
[13] D. T. Vogel and E. Kügler. The generation of artificial counter rotating vortices and the application for fan-shaped film-cooling holes. In *Proceedings of the 14th ISABE*, ISABE-Paper 99-7144, 1999.

Problem Solving Environments: Extending the Rôle of Visualization Systems

Helen Wright[1], Ken Brodlie[2], Jason Wood[2], and Jim Procter[3]

[1] Deptartment of Computer Science, University of Hull,
Hull HU6 7RX, UK
H.Wright@dcs.hull.ac.uk

[2] School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK
{kwb, jason}@scs.leeds.ac.uk

[3] now at Research School of Chemistry, Australian National University,
Canberra, ACT 0200, Australia
jimp@rsc.anu.edu.au

Abstract. Visualization systems based on the dataflow paradigm are enjoying increasing popularity in the field of scientific computation. Not only do they permit rapid construction of a display application, but they also allow the simulation to be incorporated, giving the scientist the opportunity for computational steering as well. However, if these systems are to realise their full potential as problem solving frameworks, then three key requirements of support for group working, data persistence and usability must be addressed. This paper reviews our prior work on collaborative visualization and data management and reports new developments to improve user interface flexibility. These extensions are then assessed in the context of a unifying, augmented architecture, which in turn indicates scope for future work.

1 Introduction

Advances in computing in the last ten years have brought about a significant change in the modelling and simulation of complex phenomena. Increases in computer processing power, memory, disk and graphics capacity have not only brought about a corresponding increase in the size of problem that can be attempted, but have also brought about a fundamental change in *how* such problems are tackled. Calculations which were formerly run in batch mode with their output scrutinised afterwards can now be monitored whilst in progress using graphical means, or even 'steered' by altering their input parameters according to the current visual results. One approach, exemplified in the SCIRun system [1], is to develop a purpose-built computational steering system from scratch, giving greater flexibility at the expense of more development effort. Another is to provide the tools to instrument an existing code and visualize the results (e.g. [2]). Such work has computational steering as its main focus, whilst other projects such as [3] have addressed the interworking of component-like simulation and visualization codes across heterogenous networks.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1323–1331, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The application of computational simulation to real world problems thus increasingly depends on the integration of a collection of different tools, utilised by a number of investigators having a variety of different skills. Furthermore, as projects grow in size and complexity, collaboration amongst co-workers who may be geographically separated must also be considered an issue. This, too, is recognised in [2] and has received attention in other forums [4], [5].

In this paper we review the rôle that commercially-available visualization systems may play in computational simulation, concentrating in particular on Modular Visualization Environments (MVEs). We begin by capturing an architectural model of these systems, which is then extended in three key ways.

2 Visualization Architecture and Extensions

MVEs, usefully summarised in [6], offer a variety of techniques for graphical output but without the need to program. Instead, the user selects code blocks, or modules, from a system-provided repository and joins them together using the mouse. Although widely used for post-processing simulation data, these systems are also interesting for computational steering because they allow the scientist to interact with the simulation code itself, either by incorporating it into the environment using an application programmer's interface, or by loosely-coupling the simulation and MVE. These systems implement the Filter, Map and Render pipeline model of visualization proposed by Haber and McNabb [7]. At the Filter stage, data input from disk or direct from some simulation code is sampled if dense or interpolated if sparse. In the Mapping stage, a geometrical representation is constructed, whilst at the Render stage this object is drawn and lit in order to produce the image. Modules only execute when the user varies one of their parameters or new data arrives.

In practice a number of individual modules may contribute to each of these stages and may comprise both serial and parallel codes. Coarse-grained parallelism can be exploited by distributing modules across a number of heterogeneous machines, with the environment handling user authentication and synchronisation of data flow. Alternatively, a computationally intensive code may be run on some remote supercomputer, with the results returned transparently for visualization at the workstation.

Although often the user interacts with the dataflow pipeline directly, systems also provide an end-user abstraction, or 'shrink-wrap' mode. Here, selected parameters of the simulation and visualization can be exported to a separate user interface, whilst the pipeline itself and other parameters remain private to the application developer. Figure 1 depicts a typical scenario, with a simulation code and filter elements running on a remote machine. In spite of their flexibility, MVEs also have their weaknesses. Firstly, although the dataflow pipeline may be distributed across several computers, logically it remains a single system for one user, with its interface, parameters and data flows intrinsically bound to it. Secondly, as the user changes the parameters of the calculation, so the data that flows changes. Data is ephemeral in a standard dataflow system and

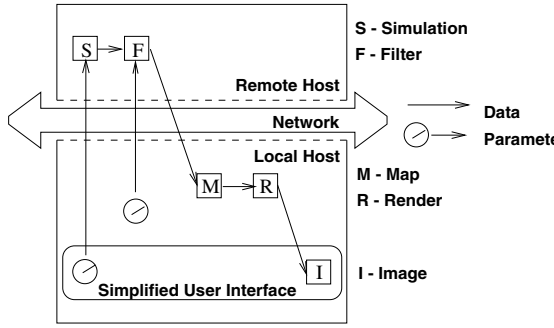


Fig. 1. MVE being used to steer a simulation running remotely

only the current state of the simulation/visualization program is ever accessible. Thirdly, complex *sequences* of actions, as may arise in computational steering, are difficult to achieve in the dataflow paradigm. Nonetheless, the general applicability of these systems and their extensibility, coupled with the infrastructure they provide for process management and distribution, warrants their consideration as generic problem solving frameworks, provided these deficiencies can be addressed.

2.1 Collaborative Working

One approach to collaborative visualization is the COVISA system described by Wood et al [8]. This system supports collaboration by allowing the selective sharing of the data, visualization parameters and pipeline building processes of an MVE. Because the MVE architecture is open, data and parameters can be captured where they flow between modules and distributed to other users. By exploiting the extensibility of the MVE we have created a set of modules, denoted by 'C' in Figure 2, that can pass data and parameters into and out of the environment. These are supported by an external server process, which is invisible to the users, for routing data between collaborators. Sharing data via modules provides a familiar paradigm and avoids the need to learn a new interface. Other modules provided allow a user to join and leave a collaborative session and to participate in collaborative map building.

This flexible approach means that users are free to choose at which points in the pipeline data and parameters are shared. Each team member works with their own copy of the visualization system, sharing data and control parameters as resource limitations and security considerations dictate. For example, in a compute-intensive application which generates a lot of data, only the geometrical representation coming from the mapping stage need be shared in order for co-workers to see the visualization (Figure 2). This configuration also means the simulation code and raw data remain private to the simulation owner, though co-workers can contribute to steering and visualization choices by exporting parameter values to their colleague's environment.

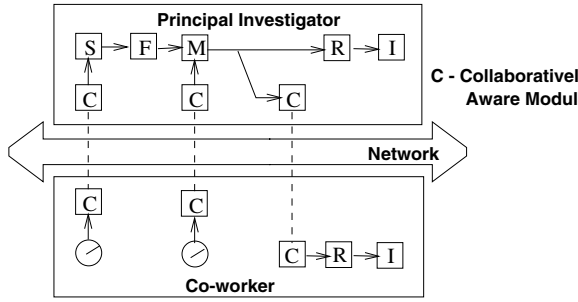


Fig. 2. Typical collaborative configuration, with shared elements reflecting resource and security constraints

COVISA is by no means the only collaborative visualization system available, for example, COVISE [9] offers collaboration by running a single instance of the base visualization system, but with multiple user interfaces, each accessing the whole pipeline. This can be achieved in COVISA by one user running all of the modules in their environment, with the others sharing control of the visualization parameters and receiving the visualized result. Another, HIGHEND [10], has multiple instances of a visualization system, one per user, each with its own user interface. The systems share synchronisation information so consistency is maintained. The equivalent in COVISA would be for each user to run the same set of modules and share control parameters. The architecture adopted here thus allows COVISA to emulate the style of collaboration offered by these other approaches, but with the key difference that users can work independently on some parts of the pipeline and collaborate over others. Whilst our implementation is for IRIS Explorer, a similar architecture has also been used to extend AVS [11].

2.2 Data Persistence

Interactive systems can bring improved insight to a problem, but the significance of any particular result is rarely appreciated at the time it is observed. More usually, hindsight plays a large part in understanding what has gone before. The need to record the progress of an investigation is thus important and has been addressed previously: the GRASPARC project [12] captured both data and parameters in the form of a tree, whose branch points signified a return to some previous simulation attempt; Mulder and van Wijk [13] have linked simulation parameters to graphical objects in the visualization, in order to see how data changes over time.

Data persistence is difficult to achieve in the dataflow model, however. Abram and Treinish [14] have proposed caching data, whilst van Liere et al [15] cite this problem as a motivating factor towards a completely new visualization architecture. Another approach by Wright et al [16] aims to be more flexible than a simple cache, but nonetheless uses a dataflow approach. Based on the GRASPARC tree idea, it includes this into the pipeline by providing an additional

module called HyperScribe. Figure 3 shows the module being used to capture simulation data for a gas turbine study, resulting from steering the ambient temperature, T. The first series of results comprises twelve distinct runs carried out at 600K to 1150K in 50K increments. As each new data set is computed the user stores it on disk by specifying an 'Add' event on the module's graphical user interface (GUI), in the process creating the circles (simulation 'snapshots') which build up the central portion of the tree from left to right. Visualization shows the optimum operating conditions to lie somewhere between 700K and 800K, so additional runs are made to fill in data between these temperatures and the extra snapshots are added to form the side branches. The entire set of results is then retrieved for visualization using 'Recall' events, again on the HyperScribe GUI.

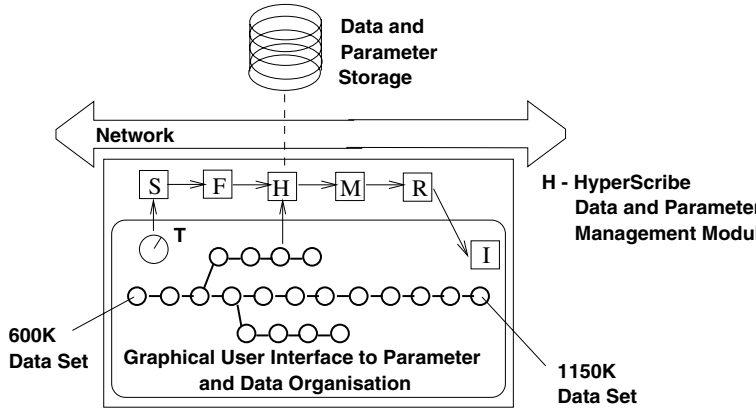


Fig. 3. HyperScribe data management

Figure 4 shows the results of tests in which users were asked to read in 12, 16 or 20 files of results and visualize them, either direct from disk or using Recall events on a tree. On average, using HyperScribe gave a 63% reduction in the time taken but, more importantly, during the 42 task instances observed, 6 errors were noted and all occurred when working with the results files directly. One reason could be a progressive loss of concentration by the user during time-consuming, repetitive tasks, in which case we might expect HyperScribe to increase nearly three-fold the size of problem that can reasonably be tackled. With planned enhancements to the HyperScribe interface, this figure could be improved still further.

2.3 Pipeline Management

Interactive systems often exhibit a repetitive element [17] and computational steering is no exception. For example, consider the following study of a chemical reaction which varies over time:

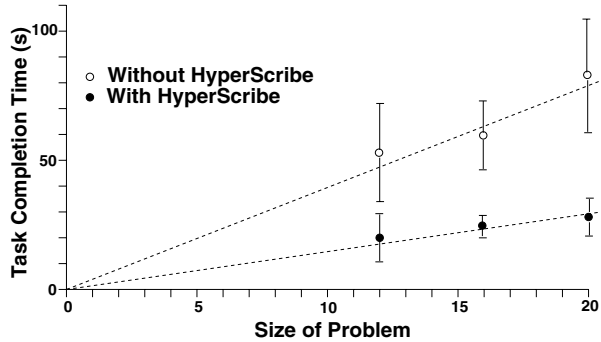


Fig. 4. Time taken to recall and visualize a 12-, 16- and 20-snapshot tree, compared with reading in the equivalent results stored as a set of files

1. Calculate new data starting from $t = 0$ and view the whole time series
2. Crop the viewed data to the portion of interest between $t = t_1$ and $t = t_2$
3. Restart the calculation from $t = t_1$, changing parameters.

In the dataflow model this requires three separate pipeline configurations which may be used repeatedly as parameters such as ambient temperature, computational tolerance and timestep are steered. Connecting and disconnecting modules manually becomes laborious and mistakes may result in data being lost. Additionally the end-user abstraction, or shrink-wrap mode, cannot be used since this hides the pipeline. Our most recent MVE extension, in the form of topology definition and management modules, seeks to support such composite activites by mapping whole configurations to a single menu item or button (Figure 5). Using the standard MVE GUI, the user constructs each pipeline just once and

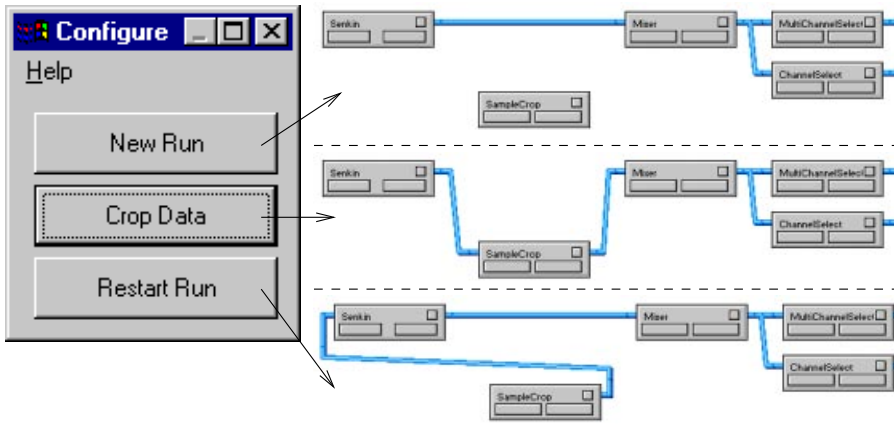


Fig. 5. Separate pipeline configurations map to a single button each

gives it a meaningful name, which is stored in a file by the topology definition module, along with the configuration. Pipeline configurations held in the file can be edited, replaced and added to for flexibility. Thereafter, pressing a particular configuration button causes the topology management module to input to the environment a script of IRIS Explorer command line instructions which changes the pipeline. The particular sequence of instructions needed is determined automatically given the module's knowledge of the current and target configurations. An application developer can choose whether to leave the pipeline available for additional manual interactions, or whether to export the configuration menu to a simplified user interface using shrink-wrap mode.

2.4 Augmented Architecture

Combining all these extensions, we can now propose an augmented architecture, Figure 6, whereby all the elements can work together. For example, HyperScribe's GUI is realised using the geometry data type, and it is a simple matter

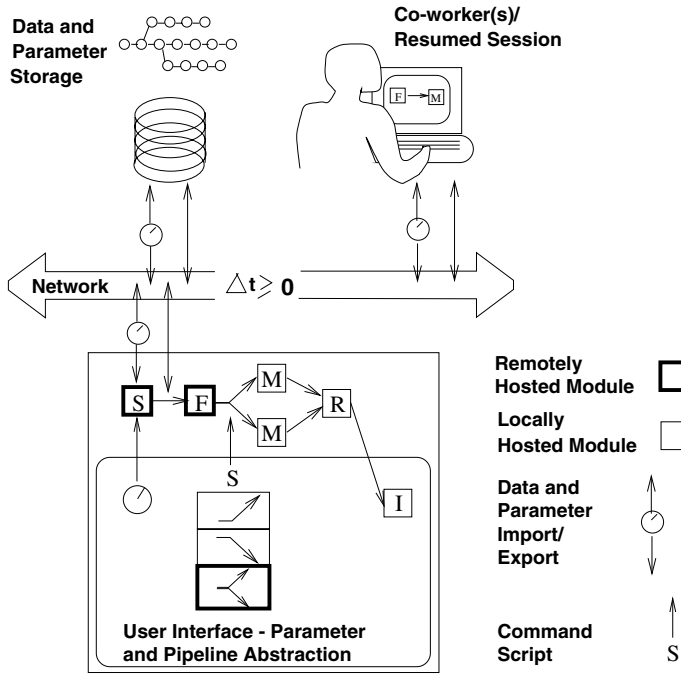


Fig. 6. Augmented MVE, with a simplified user interface giving access both to selected module parameters and different pipeline configurations. Data and parameter import and export allow different steering runs to be recorded and/or passed to co-workers, either synchronously or asynchronously

to export this to one's co-workers using a collaboratively aware module, in order to cooperate over a single database of steering results. If distant members of a team work asynchronously, anyone can pick up a database of results made earlier by a colleague by running HyperScribe remotely on that person's machine – the only collective decision needed is which particular machine to save the results on.

Our aim throughout is to work within the existing capabilities of the MVE, and this is especially important at the user interface. Here, by providing additional modules to perform the various operations, we avoid placing an additional learning burden on the user. This strategy also means our additional features can become part of a simplified interface constructed for an end-user. For example, a collaborative working application can be set up which hides the communicating modules, or different pipeline configurations can be presented simply as buttons, in a manner analogous to the abstraction of selected parameters already described.

3 Conclusions

In this paper we have reviewed past and current efforts to extend existing Modular Visualization Environments, drawing these together into an augmented architecture which demonstrates the potential for such systems to become Problem Solving Environments. A key principle of our approach has been to utilise existing MVE featues, which allows the different enhancements to interwork as required.

Improving the functionality of existing systems holds twofold benefits: firstly, we can build on work already completed to improve these; secondly, extending current systems with an established user base allows early exploitation of the ideas – IRIS Explorer alone, for example, has several hundred users. Perhaps more importantly, improving the usability of current systems allows experience gained on large-scale projects to benefit a range of other, smaller-scale endeavours. This class of applications includes those where the demand for megaflops may not be so great, but where, if a solution is to be found efficiently, the requirement for cooperative work, data and process management tools is just as real as in any large Problem Solving Environment. Thus we envisage in the coming years a growing recognition that the concept of high performance includes not only the processing power applied to some problem, but also the effectiveness of the user in solving it.

Acknowledgements

We particularly thank NAG Ltd and the UK EPSRC for funding; JP thanks Mr and Mrs G Procter for their support whilst at Leeds. Case studies, simulation code and application expertise were generously provided by BG plc, the School of Chemistry and the Department of Fuel and Energy at the University of Leeds,

UK, and Sandia National Laboratories, Livermore, USA. Finally, our thanks to the anonymous reviewers for their valuable suggestions for improvement.

References

1. C Johnson, S G Parker, C Hansen, G L Kindlmann and Y Livnat. Interactive simulation and visualization. *IEEE Computer*, **32**(12):59–65, IEEE Computer Society (1999)
2. J A Kohl and P M Papadopoulos. Computational steering and interactive visualization in distributed applications. ORNL/TM-13299, Oak Ridge National Laboratory, Oak Ridge, USA. (1999)
3. Efficient coupling of parallel applications using parallel application workspace. http://www/acl/lanl.gov/PAWS/, Los Alamos National Laboratory, California, USA. (2000)
4. K W Brodlie, D A Duce, J R Gallop and J D Wood. Distributed co-operative visualization, in Eurographics State of the Art Reports, A Augusto de Sousa and R Hopgood (editors), Eurographics98 Conference, pp27 - 50. (1998)
5. K W Brodlie and J D Wood. Volume graphics and the Internet, in M Chen, A E Kaufman and R Yagel (editors), *Volume Graphics*, pp317–331. Springer Verlag, (2000)
6. G Cameron. Modular visualization environments: Past, present and future. *Computer Graphics*, 29(2):3–4, (1995)
7. R B Haber and D A McNabb. Visualization idioms : A conceptual model for scientific visualization systems. In B Shriver G M Nielson and L J Rosenblum, editors, *Visualization in Scientific Computing*, pages 74–93. IEEE, (1990)
8. J D Wood, H Wright and K W Brodlie, Collaborative visualization, Proceedings of Visualization '97, pages 253–259. IEEE, (1997)
9. A Wierse, U Lang and R Ruhle, Architectures of Distributed Visualization Systems and their Enhancements, Eurographics Workshop on Visualization in Scientific Computing, Abingdon (1993)
10. H G Pagendarm and B Walter, A prototype of a cooperative visualization workplace for the aerodynamicist, *Computer Graphics Forum*, **12**(3):C485–C496. Eurographics, (1993)
11. D A Duce, J R Gallop, I J Johnson, K Robinson, C D Seelig and C S Cooper, Distributed Cooperative Visualization - The MANICORAL Approach, Eurographics-UK Chapter Conference, Leeds (1998)
12. K W Brodlie, A Poon, H Wright, L A Brankin, G A Banecki and A M Gay, GRASPARC: A Problem Solving Environment Integrating Computation and Visualization, Proceedings of Visualization '93. IEEE, (1993)
13. J D Mulder and J J van Wijk, 3D computational steering with parametrized graphical objects, Proceedings of Visualization '95, pages 304–311. IEEE, (1995)
14. Greg Abram and Lloyd Treinish. An extended data-flow architecture for data analysis and visualization. *Computer Graphics*, **29**(2):17–21, (1995)
15. R van Liere, J Harkes and W de Leeuw, A distributed blackboard architecture for interactive data visualization, Proceedings of Visualization '98, pages 225–231. IEEE, (1998)
16. H Wright, K W Brodlie and M Brown, The dataflow visualization pipeline as a problem solving environment, Eurographics Workshop on Scientific Visualization '96, pages 267–276. Springer-Verlag, (1996)
17. J Nielsen, Usability engineering. Academic Press, (1993)

An Architecture for Web-Based Interaction and Steering of Adaptive Parallel/Distributed Applications

Rajeev Muralidhar, Samian Kaur, and Manish Parashar

Department of Electrical and Computer Engineering and CAIP Center, Rutgers University,
94 Brett Road, Piscataway, NJ 08854.
Tel: (732) 445-5388 Fax: (732) 445-0593
{rajeevdm,samian,parashar}@caip.rutgers.edu

Abstract. This paper presents an architecture for web-based interaction and steering of parallel/distributed scientific applications. The architecture is composed of detachable thin-clients at the front-end, a network of web servers in the middle, and a control network of sensors, actuators and interaction agents at the back-end. The interaction servers enable clients to connect to, and collaboratively interact with registered applications using a conventional browser. The application control network enables sensors and actuators to be encapsulated within, and directly deployed with the computational objects. Interaction agents resident at each computational node register the interaction objects and export their interaction interfaces. An application interaction gateway manages the overall interaction through the control network of interaction agents and objects. It uses Java proxy objects that mirror computational objects to enable them to be directly accessed by the interaction web-server. The presented architecture is part of an ongoing effort to develop and deploy a web-based computational collaboratory that enables geographically distributed scientists and engineers to collaboratively monitor and control distributed applications.

1 Introduction

Simulations are playing an increasingly critical role in all areas of science and engineering. As the complexity and computational costs of these simulations grows, it has become important for the scientists and engineers to be able to monitor the progress of these simulations, and to control or steer them at runtime. The utility and cost-effectiveness of these simulations can be greatly increased by transforming the traditional batch simulations into more interactive ones. Closing the loop between the user and the simulations enables the experts to drive the discovery process by observing intermediate results, by changing parameters to lead the simulation to more interesting domains, play what-if games, detect and correct unstable situations, and terminate uninteresting runs early. Furthermore, the increased complexity and multi-disciplinary nature of these simulations necessitates a collaborative effort among multiple, usually geographically distributed scientists/engineers. As a result, collaboration-enabling tools have become critical for transforming simulations into true research modalities.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1332-1339, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Enabling seamless interaction and steering high-performance parallel/distributed applications presents many challenges. A key issue is the definition and deployment of *interaction objects* with *sensors* and *actuators* [16] that will be used to monitor and control the applications. These sensors and actuators must be co-located with the computational data-structures in order to be able to control individual application data structures. Defining these interfaces in a generic manner and deploying them in distributed environments can be non-trivial, as computational objects can span multiple processors and address spaces. The problem is further compounded in the case of adaptive applications (e.g. simulations on adaptive meshes) where computational objects can be created, deleted, modified and redistributed on the fly. Another issue is the deployment of a *control network* that interconnects these sensor and actuators so that commands and requests can be routed to the appropriate set of computational objects, and information returned can be collated and coherently presented. Finally, the interaction and steering interfaces presented by the application need to be exported so that they can be easily accessed by a group of collaborating users to monitor, analyze, and control the application.

The objective of this paper is to present the design of a web-based collaborative interaction and steering environment that addresses each of these issues. The system supports a 3-tier architecture composed of detachable thin-clients at the front-end, a network of Java interaction servers in the middle, and a control network of sensors, actuators, and interaction agents superimposed on the application data-network at the back-end. The interaction web server enables clients to connect to and collaboratively interact with registered applications using a conventional browser. Furthermore, it provides seamless access to computational and visualization servers, and simulation archives. The application control network enables sensor and actuators to be encapsulated within, and directly deployed with the computational objects. Interaction agents resident at each computational node register the interaction objects and export their interaction interfaces. These agents coordinate interactions with distributed and dynamic computational objects. The application interaction proxy manages the overall interaction through the control network of interaction agents and objects. It uses JNI [2] to create Java proxy objects that mirror the computational objects and allow them to be directly accessed by the interaction web-server. The presented research is part of DISCOVER[1], an ongoing research initiative aimed at developing a web-based interactive computational collaboratory. The current implementation enables geographically distributed clients to use the web to simultaneously connect, monitor and steer multiple applications in a collaborative fashion through a set of dedicated interaction Web Servers.

The rest of the paper is organized as follows: A brief overview of related research is presented in Section 2. Section 3 outlines the DISCOVER system architecture. Section 4 presents the design, implementation, and operation of the interaction web-server. Section 5 describes the design and implementation of control network, the application interaction substrate and its interface to the interaction server. Section 6 presents conclusions and current and future work.

---

[1] *D*istributed *I*nteractive *S*teering and *Co*llaborative *V*isualization *E*nvi*R*onment (www.caip.rutgers.edu/TASSL/Projects/DISCOVER/)

2 Related Work

Interactive systems for application run-time steering and control can be classified as follows: (1) *Event based steering systems* - These systems are oriented towards the processing of "events" that occur when pieces of code are executed. Instrumentation code is placed in the application statically at compile time at such instrumentation points. When the corresponding events occur, steering actions and decisions are taken. Systems that fall under this category include Progress [18], Magellan [16] and the CSE [5]. (2) *Systems with high-level abstractions* – In order to overcome the shortcomings of event-based steering, systems such as the *Mirror Object Steering System* (MOSS) [4] provide higher-level abstractions for steering by translating application level data structures and parameters into CORBA [13] - style objects. The DISCOVER system presented in this paper falls under this category. Key contributions of the DISCOVER architecture include (a) support for distributed dynamic interactive objects that can span multiple address spaces and can be dynamically created and destroyed, (b) a scalable control network and (c) support for web-based interaction and steering portals.

Other interactive systems include systems for interactive program construction (e.g. SCIRun [6]), systems for interactive performance optimizations (e.g. Autopilot [15]), systems for interactive application configuration and deployment (e.g. WebFlow [17], Gateway [3]). In addition to these systems, there are numerous systems that provide web-based collaborative visualization environments like *DOVE* [7], the *Web Based Collaborative Visualization* [8] system, the *NCSA Habenaro* [9] system, *Tango* [10], *CCASE* [11] and *CEV* [12].

3 DISCOVER: An Interactive Computational Collaboratory

Fig. 1 presents an architectural overview of the DISCOVER collaboratory aimed at enabling web-based interaction and steering of high-performance parallel/distributed applications. The system has a 3-tier architecture composed of detachable thin-clients at the front-end, a network of Java interaction servers in the middle, and a control network of sensors, actuators and interaction agents superimposed on the application data-network at the back-end. The front-end consists of a range of web-based client portals supporting palmtops connected through a wireless link as well high-end desktops with high-speed links. Clients can connect to a server at any time using a browser to receive information about active applications. Furthermore, they can form or join collaboration groups and can (collaboratively) interact with one or more applications based on their capabilities. The client interaction portal supports two desktops – a local desktop represents the clients' private view, while a virtual desktop presents a metaphor for the global virtual space to enable multiple users to collaborate with the aid of tools like whiteboards and chats. Application views (e.g. a plot) can be made collaborative by creating them in the virtual desktop or transferring them from the local to the virtual desktop. Session management and concurrency control is based on capabilities granted by the server. A simple locking mechanism is used to ensure that the application remains in a consistent state. DISCOVER is currently being used

to provide interaction capabilities to a number of scientific and engineering applications[2] including oil reservoir simulations, computational fluid dynamics and numerical relativity.

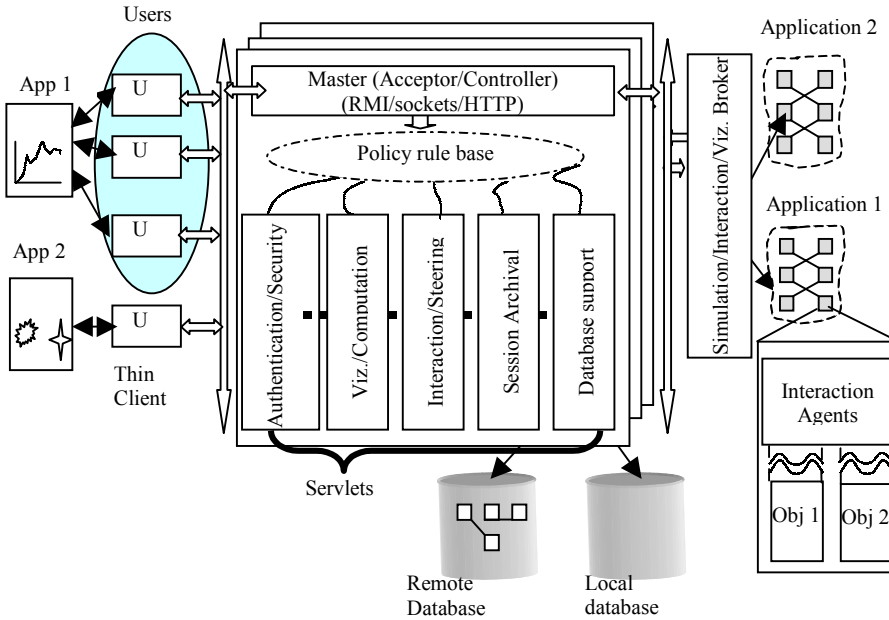


Fig. 1. Architectural Schematic of the DISCOVER Interactive Computational Collaboratory

4 Interaction and Collaboration Servers

The middle tier of the DISCOVER system consists of a network of interaction and collaboration servers aimed at providing a web-based portal to executing high-performance applications. The servers build on *Servlet* [1] technologies to add a range of specialized capabilities to traditional web-servers. A key innovation of the server architecture is the use of *reflection* to provide an extensible set of services that can be dynamically invoked in an application specific manner. The overall architecture consists of a master acceptor/controller servlet and a suite of service handler servlets including interaction and steering handler, collaboration handler, security/authentication handler, visualization handler and session archival handler and database handler. The overall architecture of the interaction server is shown in Fig.1 and the key services are described below.

[2] For example, see www.caip.rutgers.edu/TASSL/DISCOVER/ipars.html.

4.1 Collaborative Interaction and Steering

Collaborative interaction is managed by the master servlet along with the interaction and collaboration handler servlets. The master is the central hub for all communication between the client and a server. Furthermore, it coordinates all registered applications and interaction sessions. Each application, on connection, registers with this servlet, which in turn spawns a dedicated *application interaction broker*. All client requests are classified and forwarded to the corresponding broker by the interaction handler. The broker uses an application proxy to locate application objects, discover object interaction/analysis interfaces, forward queries and commands, and aggregate information from distributed objects. Application responses and updates are multicast to the interested client group by the collaboration handler. Clients can form/join/leave collaboration groups at any time. On a client connection, the master provides the incoming client with a list of registered applications. The master also accepts, parses and validates client requests and redirects them to the relevant utility servlet.

4.2 Security, Authentication, and Access Control

Security, client authentication and application access control is managed by a dedicated security and authentication handler. The current implementation supports two-level client authentication at startup; the first level is to authorize access to the server and the second level to permit access to a particular application. On successful validation of the primary authorization, the user is shown a list of the applications for which s/he has access capabilities. A second level authentication is performed for the application s/he chooses. On the client side, *digital certificates* are used to validate the server identity before the client downloads views. A *Secure Socket Layer* provides encryption for all communication between the client and the server.

To enable access control all applications are required to provide list of users and their access privileges (e.g. read, modify). The application can also provide access privileges (typically read-only) to the "world". This information is used to create access control lists (ACL). Each interaction request is then validated against the ACL before it is processed.

4.3 Application View Plug-Ins

Application information is presented to the client in the form of application *Views*. Typical views include text strings, plots, contours and iso-surfaces. Associated with each of these views is a view plug-in that can be downloaded from the server at runtime and used to present the requested view to the user. The server supports an extendible plug-in repository and allows users to extend, customize or create new views and associated plug-ins.

5 Application Control Network for Interaction and Steering

The DISCOVER control network is composed of two components: (1) Interaction Objects that encapsulate interaction sensors and actuators, and (2) a Control Network consisting of distributed Interaction Objects and Interaction Agents. These components are described below.

5.1 Sensors/Actuators and Interaction Object

Interaction objects extend application computational objects with interaction and steering capabilities, by providing them with co-located sensors and actuators. Computational objects are the data-structures/objects used by the application. Sensors enables the object to be queried whiles actuators allow it to be steered. Efficient abstractions are essential for converting computational objects to interaction objects especially when the computational objects are distributed and dynamic. In the DISCOVER system, this is achieved by deriving the computational objects from a virtual interaction base class of the DISCOVER Distributed Interaction Object library. The derived objects define a set of *View*s that they can provide and a set of *Commands* that they can accept. Interaction agents then export these views and commands to the interaction server using a simple *Interaction IDL* (Interface Definition Language). Interaction objects can be either local to a single computational node, distributed across multiple nodes, or shared between some or all of the nodes. Distributed objects have an additional *distribution* attribute that describes their layouts. DISCOVER interaction objects can be created or deleted during application execution and can migrate between computational nodes. Furthermore, a distributed interaction object can modify its distribution at any time.

In the case of applications written in non-object-oriented languages such as Fortran, application data structures are first converted into computation objects using a C++ wrapper object. These objects are then transformed to interaction objects as described above.

5.2 The Control Network and Interaction Agents

The DISCOVER control network (see Fig. 2) has a hierarchical *cellular* structure and partitions the processing nodes into a number *interaction cells.* The network is composed of (1) Discover Agents on each node, (2) Base Stations on each interaction cell and (3) an Interaction Gateway that connects to the interaction server and provides a proxy to the entire application. The number of nodes per interaction cell is programmable. The cellular control network is automatically configured at run-time using an underlying messaging environment and the available number of processors.

Discover Agents present on each node maintain run-time references to all registered interaction objects on that node. The object references can change dynamically during program execution if data is migrated to handle load balancing. The control network ensures that object references are valid and refer to consistent data. At startup, each Discover Agent exports the registered objects' interaction

information to its corresponding Base Station. Base Stations maintain a Cell Object Registry containing information about interaction objects for its interaction cell. Similarly, the Interaction Gateway maintains a Central Object Registry of interaction objects exported by all Base Stations. It is responsible for interfacing with the interaction server, delegating interaction requests to the appropriate interaction agents (Discover Agents and/or Base Stations), and collecting their responses. In the case of distributed objects, the Gateway additionally performs a *gather* operation for collating the responses arriving from the corresponding nodes.

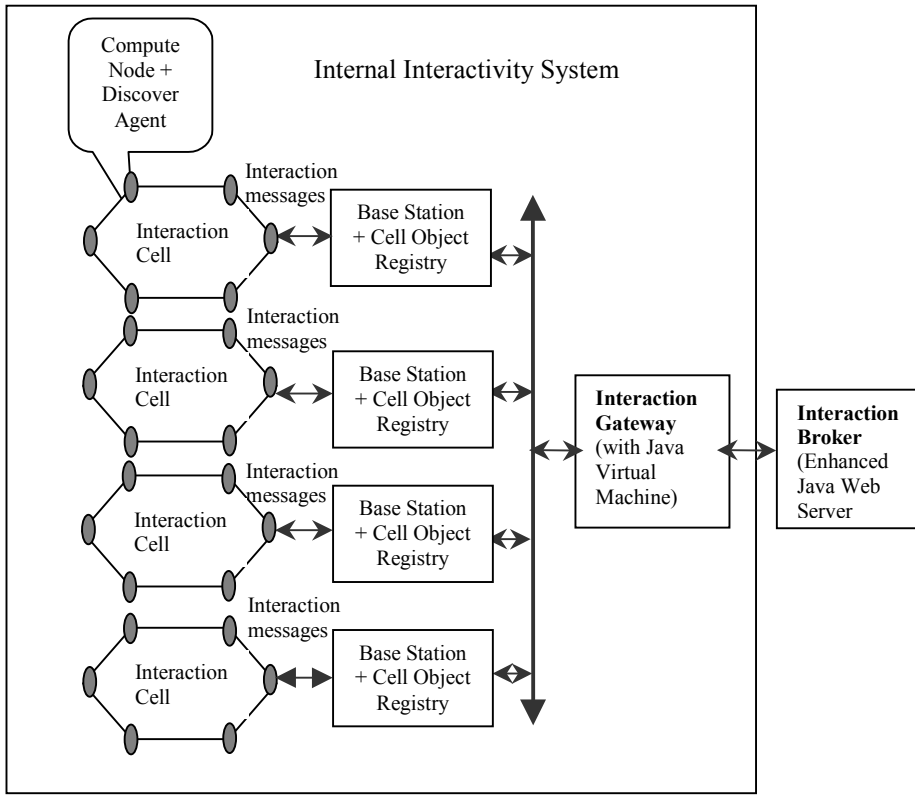


Fig. 2. DISCOVER Control Network for Application Interaction and Steering

The Interaction Gateway creates a Java mirror of each registered interaction object interface using the Java Native Interface (JNI) [2]. It uses the proxy object pattern [14] to create a placeholder for the remote (possible distributed) interaction objects (the subject) and controls access to it. This innovative feature enables interaction objects to be easily exported to an interaction server using the *Serializable* interface of Java to the interaction web server.

6 Conclusion and Future Work

This paper presented the architecture of the DISCOVER web-based computational collaboratory. The current implementation allows thin-clients at the front-end, uses enhanced Java Servlets-based interaction web servers for the middle-tier and a control network of interaction agents and interaction objects at the back-end. The interaction servers enable clients to connect to, and collaboratively interact with registered applications using a conventional browser. Current work includes implementing the Java mirroring capability using JNI at the application's interaction gateway and extending the middle tier to consist of a network of CORBA connected servers.

References

1. Hunter J.: Java Servlet Programming. 1st edition, O'Reilly, California (1998).
2. Gordon R.: Essential JNI: Java Native Interface. 1st edn. Prentice Hall, New Jersey (1998).
3. Asbury B., Fox G., Haupt T., Flurchick K.: The Gateway Project: An Interoperable Problem Solving Environments Framework for High Performance Computing. http://www.osc.edu/~kenf/theGateway.
4. Eisenhauer G., Schwan K: An Object-Based Infrastructure for Program Monitoring and Steering. 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (1998).
5. van Liere R., Harkes J., de Leeuw W: A Distributed Blackboard Architecture for Interactive Data Visualization. IEEE Viz. Conference (1998).
6. Parker S.G., Johnson S.G.: SCIRun: A scientific Programming Environment for computational steering. Proceedings of Supercomputing (1995).
7. Jain L.K.: A Distributed, Component-Based Solution for Scientific Information Management. MS Report, Oregon State University (1998).
8. Bajaj C., Cutchin S.: Web based Collaborative Visualization of Distributed and Parallel Simulation, IEEE Parallel Symposium on Visualization (1999).
9. NCSA Habenaro Home Page.: http://havefun.ncsa.uiuc.edu/habenaro. NCSA Software Development Division (1998).
10. Tango Interactive.: http://www.webwisdom.com/tangointeractive.
11. Raje R.R., Teal A., Coulson J, Yao S., Winn W., Guy III E.:CCASEE – A Collaborative Computer Assisted Software Engineering Environment. Proceedings of the International Association of Science and Technology for Development (IASTED) Conference (1997).
12. Boyles M., Raje R., Fang S.: CEV: Collaboration Environment for Visualization Using Java RMI. Proceedings of the ACM Workshop on Java for High-Performance Network Computing (1998).
13. Common Object Request Broker Architecture, http://www.omg.org.
14. Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns, Addison Wesley Professional Computing Series (1994).
15. Ribler R.R., Vetter J., Simitci H., and Reed D.: AutoPilot: Adaptive Control of Distributed Applications. 7th IEEE Symp. On High Performance Distributed Computing (1998).
16. Vetter J., Schwan K: High Performance Computational Steering of Physical Simulations. IEEE International Parallel Processing Symposium (1997).
17. Arkarsu E., Fox G., Furmanski W., Haupt T., Osdemir H., Oademir Z. O.: Building Web/Commodity base Visual Authoring Environments for Distributed Object/Component Applications – A Case Study Using NPAC WebFlow Systems.
18. Vetter J., Schwan K: Progress: A Toolkit for Interactive Program Steering. Proceedings of the 1995 International Conference on Parallel Processing (1995), 139-149.

Computational Steering in Problem Solving Environments

David Lancaster and Jeff S. Reeve

Electronics & Computer Science, University of Southampton, Southampton SO17 1BJ, U.K.
djl@ecs.soton.ac.uk

Abstract. A strong motive to build Problem Solving Environments (PSE's) is the ability to interactively steer computations. We analyse the requirements that steering puts on the architecture of such a PSE and propose a design that does not separate the development and execution parts of the PSE. We describe a prototype implementation of this design based on the standardized software infrastructure of Java Beans and CORBA that enables a high degree of steering.

1 Introduction

The desire to solve ever larger and more complex scientific and engineering problems has been one of the main driving forces in computer development. The improvement in hardware performance has been phenomenal, but the software environment in which solution methods are constructed has also evolved: notably through high level languages and scientific and mathematical libraries. Problem Solving Environments (PSE's) have been proposed [1] as a further stage in this development. A PSE is a software environment that provides support for all stages in both the development and execution of problem solving code. This paper is concerned with PSE's that incorporate, and indeed emphasise, computational steering for manipulating the course of problem solving code as it proceeds. Computational steering implicitly puts requirements of interactive responsiveness on the architecture of a PSE. In the next section we analyse some possible PSE designs in terms of these requirements. We argue that a design that does not separate the development and execution parts of the PSE allows computational steering to fit better into the control flow management and is therefore favored.

2 PSE Architecture

The commonly agreed central requirements of a PSE are to support development and execution and can be designed in two parts separately responsible for each requirement. The development part consists of a sophisticated user interface which guides the user in building a solution to his problem. The output of this development stage is a taskgraph that describes the individual tasks that must

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1340–1344, 2000.
© Springer-Verlag Berlin Heidelberg 2000

be run along with their submission order and data dependencies. Tasks consist of pieces of code and possibly parameters that tune their function. The taskgraph can be described in whatever format is convenient, a modern choice would be XML [2]. The execution part of the design takes the taskgraph and schedules it on whatever platforms are available in such a way as to satisfy all the dependency constraints encoded in the taskgraph.

This design, which we shall call the "separated design", forms the basis of many existing PSE's. It is quite apparent that this separated design limits the ability to interact with the system at run time because the taskgraph becomes immutable once it has been passed to the execution stage.

By contrast, the "combined design" integrates computational steering from the beginning and does not prejudge the kind of steering that may be necessary at run-time. The user may halt the execution, change the set of tasks and their connections, and restart the processes while preserving the state of their data. For example, a visualization module might be attached or a more accurate solver module substituted for a less accurate one. In the language of the separated design: the user is allowed to modify the taskgraph at run time.

To understand the distinction between the two designs consider the management of control flow. In general, the control flow in a PSE is determined by two factors: data flow and steering. The data flow model is intuitive in the PSE context and is made manifest in the way that the components are connected together in the graphical composer of the development part of the PSE. In the separated design, the taskgraph is an abstract representation of this data flow, possibly with hooks for a steering controller. The taskgraph is dispatched by a separate execution system and the controller manages the steering part of the control as another separate system. In the combined design, the control flow is never abstracted away from the original form as determined by the user's selection of modules and connection pattern in the graphical compositor. The components manage their own dispatching; as soon as one component has completed its task and generated its data, control passes to the next component. Computational steering fits in naturally because it is performed at the level of the graphical compositor using the same tools that were used to set up the original problem and no new control structure is needed.

3 Prototype PSE

Software developments for commodity markets over the last few years have provided us with the some basic tools that can be used to construct PSE's. There are strong arguments to use these software layers in designing software for scientific and engineering purposes, notably: they are cheap (often with free implementations), widely available and most importantly they are standardized.

Our prototype PSE uses JavaBeans [3] and CORBA [4]. These are appropriate for a prototype that is intended to be used with medium sized platforms such as clusters [5] small shared memory machines and individual workstations. A more highly featured PSE that was intended to be used with more substantial

computing resources would also need software layers that connect to queuing systems and databases.

JavaBeans fulfill the need for a sophisticated user interface that allows graphical programming by connecting computational components. CORBA provides a platform independent way of distributing the computations performed by the PSE so as to take advantage of high performance machines and software implementations.

To illustrate the use of the PSE and to make the following discussion more concrete we have implemented some simple components that perform operations that diagonalize a matrix [6]. These components include one to generate the matrix and components Householder and QL that manipulate the form of the matrix.

The beans representing these components have interfaces described in the BeanInfo class that prescribe how they can be wired together in the builder tool. Besides listing any parameters, this class lists the type and form of the inputs and outputs that the component expects. The JavaBean code is essentially limited to describing these interfaces and the CORBA client code needed to call the computational module residing on a server. For this reason we call these beans "hollow". In order to avoid unnecessary data movement and thereby improve performance, the form of the inputs/outputs are not explicit matrices, but are CORBA references to data objects that contain the matrices.

The CORBA part of the prototype development requires an IDL for the system including the data objects for the matrices. This IDL is too long and detailed to include in a paper but is available on `http://gather.ecs.soton.ac.uk/PSE/Docs`. The CORBA servers have evolved since the beginning of this project and now provide facilities to manage the life-cycle of computational components.

We employ a performance enhancement that makes non-blocking CORBA calls to hide latency following a standard CORBA technique. Consider for example the Householder component which accepts a symmetric matrix and generates a tridiagonal matrix. When this bean starts to run, it first contacts the scheduler which provides an object reference on some remote machine to the computational class that implements the householder transformation. The bean then uses this reference to submit the object reference of the symmetric matrix data class and control then passes back to the bean which then waits. Meanwhile, the Householder object uses the matrix reference to transfer the matrix data to its local address space and then proceeds with the computation, finally creating a new matrix object in which to store the resulting tridiagonal matrix. The object reference to this tridiagonal matrix object is returned to the bean indicating completion of the job.

For the purposes of this work we implemented a scheduler that allows several versions of each component to exist on different machines and selects which one to use on request. This scheduler is dumb in the sense that the algorithm for selection is extremely simple, being either random or based on prior user choices. It does however provide the same kind of interface expected in a more sophisticated version. CORBA initialization employs a nameserver which is used

to register servers that are started by hand on whatever remote machines are available. The scheduler contacts the nameserver to obtain all the information about which components are available on which machines.

The control flow information has its origin in the act of wiring JavaBeans together and remains encapsulated in the builder tool. It is never abstracted into a taskgraph. As soon as one component has completed its task and generated its data, control passes to the next bean which dispatches a request to the CORBA object that implements the computational work of the component. The flow of control between one component and another takes place using the standard bean event mechanism and PropertyChangeEvents trigger this flow. For example, when the householder bean in the example above, obtains the reference to the tridiagonal matrix object, it fires a PropertyChangeEvent that is picked up by the next bean, in this case the QL bean, that has been wired to listen for such events.

The fact that control flow resides entirely in the standard beans on the builder tool is the essential aspect that realizes the combined design and allows the computational steering freedom that we have emphasised throughout.

4 Conclusion

We have analysed the design of a PSE intended to provide good support for computational steering. Then by implementing a prototype incorporating this design we have shown that JavaBeans and CORBA provide an appropriate software basis for small/medium sized PSEs, allowing a high degree of run-time freedom to steer the computation.

We have in mind several improvements to the prototype implementation which would make this PSE a more useful and robust tool. To extend the type of steering that can be done, we would like to construct our own builder tool that would allow beans to be deleted or substituted at run time. To allow the freer introduction and sharing of new components they should be described in some language such as XML [7] along with tools that use this description to create most of the beans and BeanInfo classes as well as providing a skeleton for the computational part of the code. The other major area for improvement is in the scheduler, which should monitor load and provide more sophisticated selections on the basis of some knowledge of the intended taskgraph. However, a significant conclusion of this paper is that the scheduler should not be allowed to gain control of dispatching – or the freedoms of computational steering will be forfeited.

Acknowledgments

DL would like to acknowledge discussions with Peter Lockey and Matthew Shields. This work was supported by a UK EPSRC grant entitled, *Problem Solving Environments for Large Scale Simulations*.

References

1. E. Gallopoulos, E. Houstis and J.R. Rice. *Problem Solving Environments for Computational Science*, IEEE Comput. Sci. Eng., **1**, 11-23, 1994.
   E. Gallopoulos, E. Houstis and J.R. Rice. *Workshop on Problem Solving Environments: Findings and Recommendations*, ACM Comp. Surv., **27**, 277-279, 1995.
2. World Wide Web Consortium, *Extensible Markup Language (XML)*
   `http://www.w3.org/XML/`
3. `http://java.sun.com/beans/`
4. The CORBA specification is controlled by the Object Management Group
   `http://www.omg.org/`
5. D. Ridge, D. Becker, P. Merkey and T. Sterling. *Beowulf: Harnessing the Power of Parallelism in a Pile-Of-PC's*, Proc. 1997 IEE Aerospace Conference.
   See the Beowulf Project page at CESDIS:
   `http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html`
6. D. Lancaster and J.S. Reeve. *A Problem Solving Environment Based on Commodity Software*, Proc. HPCN2000. Lecture Notes in Computer Science **1823**, Ed. M. Bubak, H. Afsarmanesh, R. Williams and B. Hertzberger.
7. O.F. Rana, M. Li, D.W. Walker and M.Shields. *An XML based Component Model for Generating Scientific Applications and Performing Large Scale Simulations in a Meta-Computing Environment*, available from:
   `http://www.cs.cf.ac.uk/PSEweb/`

Implementing Problem Solving Environments for Computational Science

Omer F. Rana[1], Maozhen Li[1], Matthew S. Shields[1], David W. Walker[2], and David Golby[3]

[1] Department of Computer Science, University of Wales, Cardiff, POBox 916, Cardiff CF24 3XF, UK
[2] Computational Sciences Section, Oak Ridge National Laboratory, PO Box 2008, Oak Ridge TN 37831-6367, USA
[3] Department of Mathematical Modelling, British Aerospace Systems, Sowerby Research Center, PO Box 5, Filton, Bristol, BS34 7QW, UK

Abstract. A Problem Solving Environment (PSE) should aim to hide implementation and systems details from application developers, to enable a scientist or engineer to concentrate on the science. A PSE is, by definition, problem domain specific, but the infrastructure for a PSE can be problem domain independent. A domain independent infrastructure for a PSE is described, followed by two application dependent PSEs for Molecular Dynamics and Boundary Element codes that make use of our generic PSE infrastructure.

1 Introduction

A Problem Solving Environment (PSE) is a complete, integrated computing environment for composing, compiling, and running applications in a specific area [3]. PSEs have been available for several years for certain specific domains, but most of these have supported different phases of application development, and cannot be used cooperatively to improve a scientists' productivity, primarily due to the lack of a framework for tool integration and ease-of-use considerations. Extensions to current scientific programs such as Matlab, Maple, and Mathematica are particular pertinent examples of this scenario. Developing extensions to such environments enables the reuse of existing code, but may severely restrict the ability to integrate routines that are developed in other ways or using other applications. Multi-Matlab [4] is an example of one such extension for parallel computing platforms.

A PSE must contain: (1) application development tools that enable an end user to construct new applications, or integrate libraries from existing applications, (2) development tools that enable the execution of the application on a set of resources. In this definition, a PSE must include resource management tools, in addition to application construction tools, albeit in an integrated way. Component based implementation technologies provide a useful way of achieving this objective, and have been the focus of research in PSE infrastructure. Based on the types of tools supported within a PSE, we can identify two types

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1345–1349, 2000.
© Springer-Verlag Berlin Heidelberg 2000

of users: (1) application scientists/engineers interested primarily in using the PSE to solve a particular problem (or domain of problems), (2) programmers and software vendors who contribute components to help achieve the objectives of the category (1) users. The PSE infrastructure must support both types of users, and enable integration of third party products, in addition to application specific libraries.

Many of these requirements are quite ambitious, and existing PSE projects handle them to varying extents. The component paradigm has proven to be a useful abstraction, and has been adopted by many research projects, making use of existing technologies such as CORBA, JavaBeans and DCOM/COM. Although a useful abstraction for integration, performance issues when wrapping legacy codes in Fortran as components have not been addressed adequately. Automatic wrapper generators for legacy codes that can operate at varying degrees of granularity, and can wrap the entire code or sub-routines within codes automatically, are still not available. Part of the problem arises from translating data types between different implementation languages (such as complex numbers in Fortran), whereas other problems are related to software engineering support for translating monolithic codes into a class hierarchy. Existing tools such as Fortran to Java translators cannot adequately handle these specialised data types, and are inadequate for translating large application codes, such as the Lennard-Jones molecular dynamics application discussed in section 2. For PSE infrastructure developers, integrating application codes provides one goal, the other being the resource management infrastructure to execute these codes. The second of these can involve workstation clusters, or tightly coupled parallel machines. We therefore see a distinction between these two tiers of a PSE, (1) a component composition environment, (2) a resource management system. A loose coupling between these two aspects of a PSE will be useful where third party resource managers are being used, whereas a strong coupling is essential for computational steering or interactive simulations.

2 Applications Using the PSE Infrastructure

We describe two applications that makes use of our generic component based PSE infrastructure described in [8].

Molecular Dynamics: The molecular dynamics application models a Lennard-Jones fluid. It is written in C, and was initially wrapped as a single CORBA object. A Java interface component is combined with a component that wraps the executable code, enabling input from the user to be streamed to the executable, and subsequently to display output results to the user. The CORBA object makes use of an MPI runtime internally, and is distributed over a cluster of workstations. The intra-communication within the object is achieved via the MPI runtime, while the interaction between the client and the object is via the ORB. An application developer does not need to know the details of how the code works, or how to distribute it over a workstation cluster. The wrapper therefore provides the abstraction needed to hide implementation details from

an application user, requiring the developer to automatically handle exceptions generated from the CORBA and MPI system. The molecular dynamics code was subsequently sub-divided into four CORBA objects, to improve re-usability, and included: (1) an `Initialization` object to calculate particle velocities and starting positions, (2) a `Moveout` object to handle communication arising from particle movement and "ghost" regions, (3) a `Force` object which calculates the force between molecules, and constitutes the main computational part of the code, (4) an `Output` object that generated simulation results for each time step. These four objects were managed by a `Controller` object, which coordinated the other four objects. A user now has more control over component interaction, but still does not know how the MPI runtime is initialised and managed within each object. The application developer can construct a molecular dynamics application by connecting components together. Each component is self-documenting based on its XML interface, and component properties can be investigated using an Expert Advisor [7].

BE2D: The BE2D code is a 2D boundary element simulation code for the analysis of electromagnetic wave scattering. The main inputs to the program are a closed 2D contour and a control file defining the characteristics of the incident wave. The original code was written in Fortran. To make the code easier to use within the PSE and also to provide runtime speed comparisons it was decided to convert the code into Java. The use of Fortran to Java converters was considered but abandoned fairly quickly because of the lack of support for complex numbers. Complex numbers caused a number of problems in converting the code, not least because of the fact that they are primitive types in Fortran, but are not directly supported in the standard Java API. A third party implementation of complex numbers for Java was found and used, from Visual Numerics called JNL [5]. Complex number arithmetic proved to be more than just a straight translation from the Fortran code to its Java equivalent. As complex numbers in Java are objects, and Java does not yet support operator overloading, the complex number objects have to use their operator methods, add, multiply, subtract and divide. These methods are either unary or binary, taking a single input parameter which is used to modify the object calling the method or taking two input parameters on a static class method that creates a new result without any modification side effects. Thus calculations involving more than an arbitrary number of parameters become far more complicated in Java than their Fortran equivalents. The converted BE2D was subsequently used as a single component within the PSE. The solver is combined with a graph generator (JChart [6]) as a third party component. The output generated from the code is illustrated in figure 1. If the complete BE2D solver is wrapped as a CORBA object, comprising a single Fortran executable unit, then the interface contains the name of the solver, and a single input file. If the data is to be streamed to the solver, a component that can convert a file (local or at a URL) into a stream is placed before the component.

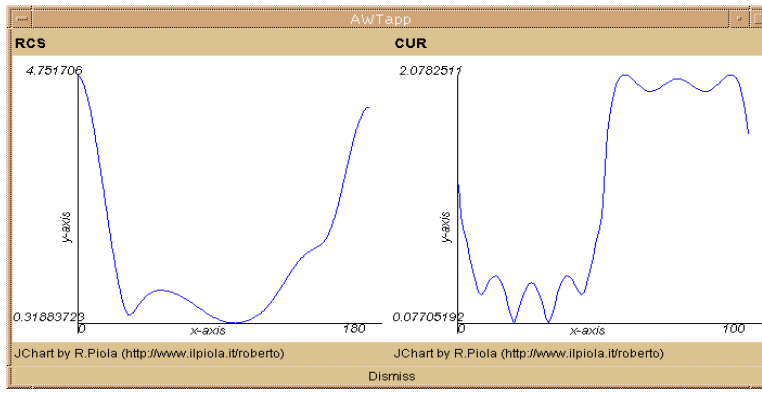


Fig. 1. *Output of the BE2D solver*

3 Conclusion and Future Work

Current advances in networking and distributed object technologies provide the infrastructure to support the creation of general purpose PSEs. The emphasis in developing PSEs should be on creating an environment that is easy to use, and intuitive for an application/domain expert. A computational scientist should therefore not have to configure hardware resources in order to undertake molecular dynamics research, for instance. An important lesson learnt from our work is the necessity to understand the impact of specialised data structures on the usage and performance of legacy codes, when these are wrapped as components within a PSE. The Java programming language can be useful in integrating various components, but it still suffers from the absence of data structures that are essential for scientific codes. Current efforts are under way within the JavaGrande forum [1] to address some of these issues, and the result of these efforts will be significant to future PSE infrastructure.

References

[1] The JavaGrande Forum. See web site at: `http://www.javagrande.org/`.
[2] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Int. Journal of Supercomputing Applications*, 11(2), 1997.
[3] E. Gallopoulos, E. N. Houstis, and J. R. Rice. Computer as Thinker/Doer :Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering*, 1(2), 1994.
[4] Vijay Menon and Anne E. Trefethen. MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing. *Proceedings of SuperComputing97*, 1997.
[5] Visual Numerics. JNL: A numerical library for Java. See web site at: `http://www.vni.com/products/wpd/jnl/`.
[6] Roberto Piola. The JChart package. See web site at: `http://www.ilpiola.it/roberto/jchart/`.

[7] M. Shields, O. F. Rana,D. W. Walker, M. Li, and D. Golby, A Java/CORBA based Visual Program Composition Environment for PSEs, Concurrency: Practice and Experience (in press), 2000

[8] D. Walker, M. Li, O. Rana, M. Shields, and Y. Huang. The Software Architecture of a Distributed Problem Solving Environment. Technical report, Oak Ridge National Laboratory, Computer Science and Mathematics Division, PO Box 2008, Oak Ridge, TN 37831, USA, December 1999. Research report no. ORNL/TM-1999/321.

Pseudovectorization, SMP, and Message Passing on the Hitachi SR8000-F1

Matthias Brehm, Reinhold Bader, Helmut Heller, and Ralf Ebner

Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften,
Barer Straße 21, 80333 München,
Germany
{Brehm, Bader, Heller, Ebner}@lrz.de
http://www.lrz-muenchen.de/services/compute/hlrb

Abstract. The Leibniz-Rechenzentrum in Munich has started operating a 112-node Hitachi SR8000-F1 with a peak performance of 1.3 Teraflops in the second quarter of 2000, the fastest computer in Europe. In order to make use of the full memory bandwidth and hence to obtain a significant fraction of the peak performance for memory intensive applications, the compilers offer preload and prefetch optimization strategies to pipeline load/store operations, as well as automatic parallelization across the 8 processors contained in every node. The nodes are connected by a conflict-free crossbar, enabling efficient communication via standard message-passing interfaces. An overview of the innovative architectural concepts is given. We demonstrate to which extent the capabilities of the compiler to automatically pseudovectorize/parallelize typical application code are sufficient to produce well-performing code.

1 Aiming for Top Level Computing

In the first quarter of 2000, the Leibniz-Rechenzentrum (LRZ) Munich installed a Hitachi SR8000-F1 intended to serve as Top-Level Compute Server in Bavaria (the German acronym HLRB will be used in the following); this machine will be again enlarged by approximately half its present computing power in a second installation phase in 2Q2002. In installation Phase I the system consists of 112 nodes. Each pseudo-vector node contains 9 CPUs, 8 of which are available for computational tasks, and 8 Gigabytes memory which is accessible from the processors in a shared-memory model. The CPUs are similar to the IBM Power Architecture, with proprietary extensions added by Hitachi (cf. Section 2.1). Since the processors are operated at a frequency of 375 MHz and 4 floating point operations can be executed per cycle, each pseudo-vector node yields 12 GFlops peak performance. Thus the HLRB has a peak performance of 1.344 TFlops. The ninth processor on each node is needed as a service processor. The nodes of the SR8000-F1 are inter-connected via a three-dimensional crossbar with a bi-directional bandwidth of 2x950 MB/s between two nodes and a hardware latency of about 5 microseconds. Further details of the HLRB hardware are given in Tables 1 and 2 below.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1351–1361, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The LINPACK performance value of the HLRB is 1035 GFlops and a sustained application performance has been measured to 450 GFlops. Hence, LRZ is currently operating the fastest computer within Europe. Usage of the HLRB will be open to German research projects which need high sustained performance and are presently not feasible on any other existing computing platform. Resources will be allocated to individual projects after a peer review process. Vectorizable Codes will be preferred, however the SR8000 architecture is sufficiently flexible that the system may be used in MPP mode as well.

2 The Innovative Architecture of the SR8000-F1

The architecture of the SR8000-F1 allows the usage of the vector programming paradigm and the scalar SMP-Cluster programming paradigm on the same machine. This is achieved by combining the superscalar RISC CPUs into a virtual vector CPU. In a traditional vector CPU the vectorized operations are executed by a vector pipe which delivers one or more memory references per cycle to the CPU. On the Hitachi SR8000-F1 the vectorizable operations are distributed among the 8 effectively usable CPUs of a node ("COMPAS", COoperative Micro-Processors in single Address Space); furthermore in case of memory-bound computing, specific memory references can be loaded into the registers or the caches some time ahead of actual use ("PVP", Pseudo-Vector-Processing). These two properties of the SR8000-F1 nodes especially contribute to the high efficiency obtained in comparison to other RISC systems.

	Phase I Configuration 1Q2000	Phase II Configuration 2Q2002
Number of SMP-Nodes	112	168
CPUs per Node	8 (+1 Service)	8 (+ 1 Service)
Number of Processors	112*8 = 896	168*8 = 1344
Peak Performance per CPU	1.5 GFlop/s	1.5 GFlop/s
Peak Performance per Node	12 GFlop/s	12 GFlop/s
Peak Performance SR8000	1344 GFlop/s	2016 GFlop/s
LINPACK Performance of the whole System	1035 GFlop/s	to be measured
Performance from main memory (most unfavourable case)	163.5 GFlop/s	244 GFlop/s
Memory per Node	8 GByte	8 GByte
Memory of total system	928 GBytes	1344 GBytes
Aggregated Disk Storage	7.4 TBytes	10 TBytes
Bidirectional Communication bandwidth using MPI	2x950 MByte/s (Hardware: 1 GByte/s)	2x950 MByte/s (Hardware: 1 GByte/s)

Table 1. Hardware Overview of LRZ's HLRB.

Processor and Memory Characteristics	
Frequency and Processor Cycle	375 MHz (2.67 nanoseconds)
Maximum Number of Operations per Cycle	4
Number of Floating Point Registers	160 (Global: 32, Slide: 128)
Number of Integer Registers	32
Data Cache Size	128 KB (write through, 4-way set associative, direct mapped)
DCache line size	128 Bytes
DCache bandwidth to registers	32 Bytes / cycle
Memory Frequency and mem-cycle	250 MHz (4 ns)
Maximum Number of Loads from Memory	16 Bytes / mem-cycle
Bandwidth to Memory per Processor	4 GB/s (32 GB/s peak for 8-processor node)

Table 2. Properties of the processor and memory system used by Hitachi in the Phase I installation. *It has not yet been decided whether a more advanced CPU model will be used in the Phase II upgrade.*

2.1 Pseudo-Vector-Processing (PVP)

Hitachi's extensions to the IBM POWER instruction set, which improve the memory bandwidth, alleviate the memory bottleneck, which is the main deficit of RISC-based High Performance Computing. This property, called Pseudo-Vector Processing by Hitachi, may be used by the compiler to obtain data either directly from memory via preload or via prefetch through the cache, depending on how the memory references are organized (see Fig. 1 below). The concept of PVP may be illustrated by the following example loop:

```
DO I=1,N
   A(I) = B(I) + C(I)
ENDDO
```

Using prefetch operations, which may be overlapped with the floating point operations, one obtains the sequence shown in the right part of Figure 2. Prefetch is not very efficient when the main memory is accessed non-contiguously because the prefetched cache line may contain unnecessary data. To improve this situation the preload mechanism was implemented. Preload transfers element data directly to the registers as illustrated in Figure 3. The physical registers are mapped to logical addresses via a sliding window technique. A special instruction (sliding window step) is used to update the slide window base value which is held in a special purpose register.

2.2 Cooperative Micro Processors in Single Address Space (COMPAS)

COMPAS enables the automatic distribution of computational work of a loop among the 8 CPUs of an SMP node by the compiler (autoparallelization) and the accompanying hardware support for synchronization. COMPAS may also

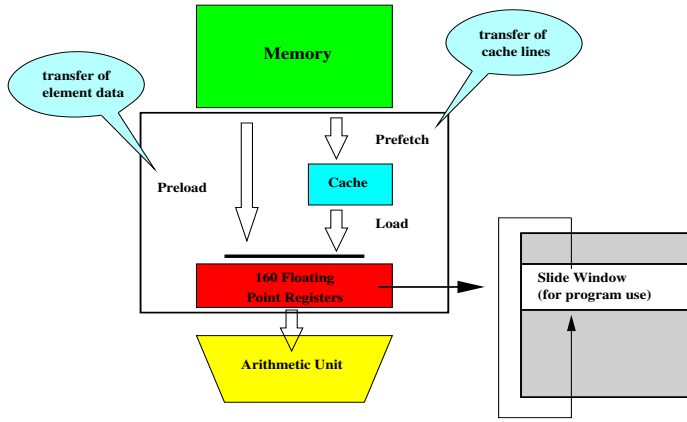


Fig. 1. Pseudo-vectorization: Prefetch and Preload.

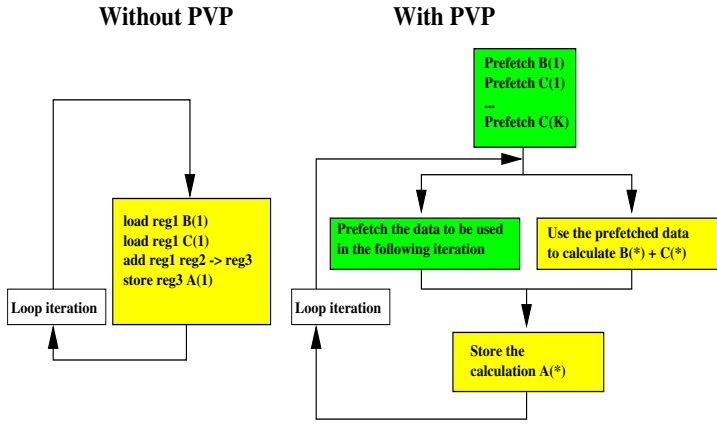


Fig. 2. Loop structure without and with PVP Prefetch.

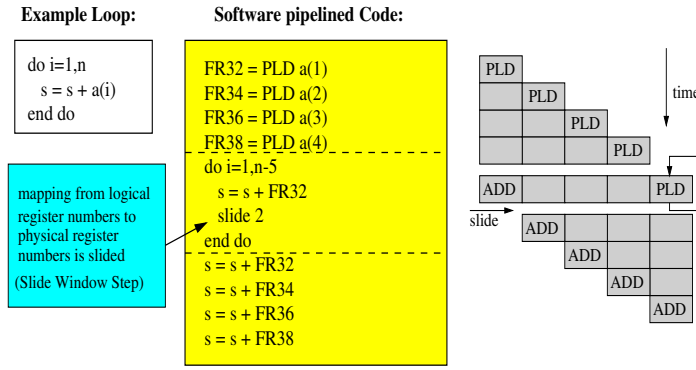


Fig. 3. Loop Structure with PVP Preload.

be utilized by codes which use the nodes as 8-way SMP-nodes via OpenMP, since version 1.0 of the OpenMP standard is implemented as part of the Hitachi Fortran Compiler.

3 Benchmark Results and Principles for Code Optimization

The most important criterion for evaluation of the offered machines was not the peak performance but the actually obtained "sustained" performance for a suite of application benchmarks. Furthermore, several additional tests were performed to obtain a measure of how the hardware performs in the least favorable situations or to evaluate scaling of MPI codes to the largest possible problem size. The examples discussed in the following subsections will also provide insights on the principles of optimizing code for the SR8000.

3.1 Memory Throughput

STREAM Benchmark This program (written by John D. McCalpin) is used to evaluate the memory bandwidth of a node. The following loops are performed:

Copy	Scale	Add	Triad
DO J=1,N C(J)=A(J) END DO	DO J=1,N B(J)=S*C(J) END DO	DO J=1,N C(J)=A(J)+B(J) END DO	DO J=1,N A(J)=B(J)+S*C(J) END DO

The vector length N used in this case was 19,121,111, corresponding to a memory usage of 437 MB for the program, and ensuring that it is really the transfer from/to memory that is measured. The memory bandwidth is plotted in Figure 4 for all types of loops indicated above. A comparison with other vendor's hardware is provided in Table 3. The machine balance is obtained as

Platform	Bandwidth (MB/s)	Peak Performance (MFlops)	Machine Balance (Flop/Word)	Remarks
SR8000-F1	22311	12000	4.3	8 Processors, 375 MHz
R12000 (SGI)	811	2400	23.7	4 Processors, 300 MHz
Pentium III	396	500	10.1	1 Processor, 500 MHz
Cray C90	103812	15360	1.2	16 Processors
NEX SX-5	583069	128000	1.8	16 Processors
IBM Nighthawk1	3872	7104	14.7	8 Processors, 222 MHz

Table 3. Overview of Triad memory bandwidth for various platforms.

the quotient of peak performance and bandwidth, the latter being expressed in 8-Byte words; lower numbers are better, as far as memory-intensive (out-of-cache) computing is concerned. Typically, RISC-based systems are at least an order of

magnitude worse than the specialized vector processors. However, Hitachi has nearly managed to bridge this gap by having a better ratio of memory cycle to processor cycle to start with, as well as being able to access memory from all processors in the SMP node simultaneously without too high losses: Of the 32 GBytes/s per node naively calculated from the single-processor bandwidth of at least 22 GByte/s can actually be obtained. In order to see how the memory bandwidth scales with the number of processors used, an OpenMP parallelized version of the STREAM benchmark was run. Figure 4 shows the efficiency

$$E(n) = \frac{\text{measured Bandwidth}(n)}{n \cdot \text{peak Bandwidth}(1)}$$

as a function of the number of threads for the various loop types. The peak bandwidth for 1 processor is assumed to be 4 GB/s (cf. Table 2). One ob-

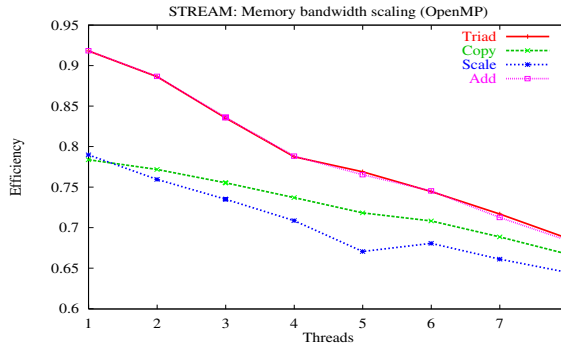


Fig. 4. Memory band-width as measured in STREAM benchmark

serves a degradation for increasing numbers of threads. The differences between Triad/Add and the other two tests are accounted for by a difference between Load and Store: Triad/Add involves 2 Loads and 1 Store, while Scale/Copy has only 1 Load and 1 Store. However it is not entirely clear why there is a measurable difference between Scale and Copy. It must be remarked that other RISC SMP machines degrade far more than the SR8000 node: On 8-way systems one can expect at most 40% of peak bandwidth. In the top ten list kept at `http://www.cs.virginia.edu/stream/top10/Bandwidth.html` the SR8000-F1 (as well as its predecessor) would be ranked at position 7.

Parallelization and Pseudo-vectorization for Triads Performing Triads for variable vector length yields information not only about the memory throughput, but of the complete register/cache/memory system. Furthermore, some of the compiler's capabilities to automatically parallelize and pseudovectorize code are investigated. Figure 5 shows the per-formance of the (3 load + 1 store, 2 operation) triad

```
DO I=1,N
   A(I) = B(I)*C(I) + D(I)
END DO
```

as a function of vector length for the four execution modes possible on the SR8000: 1) COMPAS-parallel and pseudo-vectorized, 2) COMPAS-parallel without pseudo-vectorization, 3) Non-parallel, but pseudo-vectorized, 4) Neither parallel nor pseudo-vectorized. For these measurements, the desired effect was obtained within a single program unit by inserting appropriate compiler directives. Looking at case 3) first, one observes uniformly high performance of up to 490 MFlops until the Level-1 Cache is exhausted at approximately $n = 4000$. After that, performance is governed by the memory throughput, where PVP achieves around 230 MFlops in the $n \to \infty$ limit. Using 8 IPs in parallel allows for eightfold cache usage, hence the vector-like range ends at n=30000, where a performance of 3360 MFlops is reached, the 6.86-fold of what is obtained on a single IP. The vector-like characteristic of the performance for small n is due to the domination of COMPAS startup times for short vector lengths. The size of the cache performance window strongly depends on the particular loop kernel. Very fat loop kernels may lead to register spill and hence may require to be split up, too thin kernels are fused or suitably unrolled. All of this is automatically performed by the compiler.

The $n \to \infty$ performance achieved here is 1410 MFlops, which is the 6.13-fold value of a single IP. *Hence, due to PVP the triad achieves more than 40% of its maximum performance even when leaving the cache performance window, while without PVP one obtains the RISC-typical value of around 7 %.* This is of very high importance for memory intensive computing tasks in technics and science.

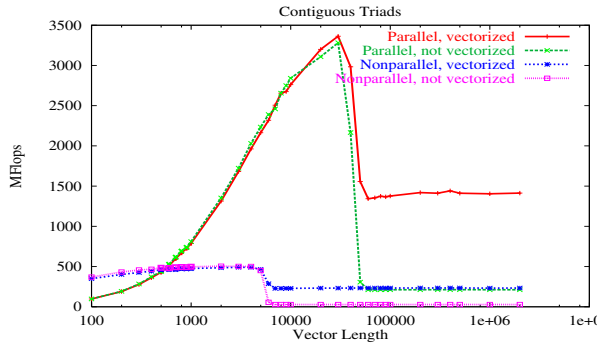


Fig. 5. Performance of Contiguous Triads as function of vector length.

3.2 Scalability of MPI Programs

Part of the LRZ benchmark suite was concerned with scalability studies. The programs

- FT (Fast Fourier Transform from NAS Parallel Benchmarks)
- MG (basic Multigrid Algorithm from NAS Parallel Benchmarks)
- HRD (ScaLAPACK call for Hessenberg transformation)

were performed on the SR8000-F1, the results of which are presented in the following. The Class C 512x512x512 three dimensional Fast Fourier Transform and the Class C Multigrid Test were performed in COMPAS mode on an increasing number of nodes. The HRD Test was performed with a fixed number of nodes but varying matrix sizes. The tests generally showed high scalability and very good performance. From the programmers point of view, it is advantageous that one has to deal only with a relatively small number of nodes instead of the eightfold higher number of processors.

Benchmark	Nodes	Processors	Performance (GFlops)	MPI Efficiency (relative to 4 nodes)
FT	4	32	15.5	1.00
	8	64	29.5	0.95
	16	128	56.9	0.92
	32	256	109.8	0.89
MG	4	32	14.8	1.00
	8	64	27.7	0.93
	16	128	46.9	0.79
	32	256	72.3	0.61

Benchmark	Nodes	Processors	Performance	Matrix Size
HRD	32	256	11.8	1000 x 1000
	32	256	52.8	10000 x 10000
	32	256	134.0	30000 x 30000

3.3 Case Studies for the Hybrid (COMPAS/OpenMP + MPI) Programming Paradigm

Parallel Vector Times Matrix As an example for the hybrid programming model, we are going to look at a simple implementation of vector-matrix multiplication. Using MPI for the distributed memory version, there are only few changes from the serial version. The structure of the code is shown in Figure 6 (for the pure MPI version the OpenMP directives are just ignored). The performance for matrix size n=10000 and n=40000 is given in Figure 7. It is obvious that the hybrid version performs much better than the pure MPI version. We first thought that this was due to the reduced amount of MPI communication. However, further examination showed that not only the communication part had shorter execution time, but also the computational part. The latter effect is caused by the increased vector length in the algorithm. For an algorithm with higher ratio of MPI-communication to computation the differences will be even more marked.

Parallel Matrix Multiplication Parallelization of Matrix Multiplication via MPI uses a ringcast scheme for matrix blocks, where the block size is chosen

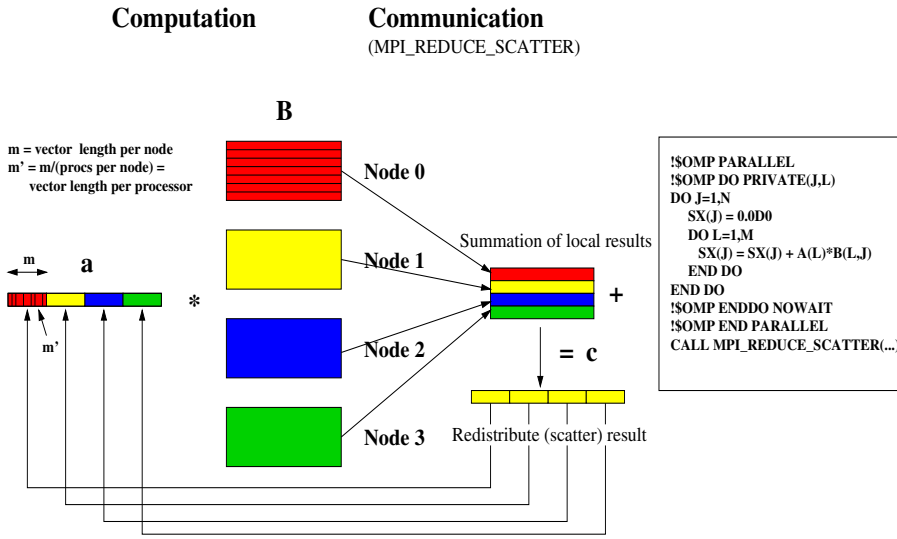


Fig. 6. Code Structure for parallel vector times matrix multiplication.

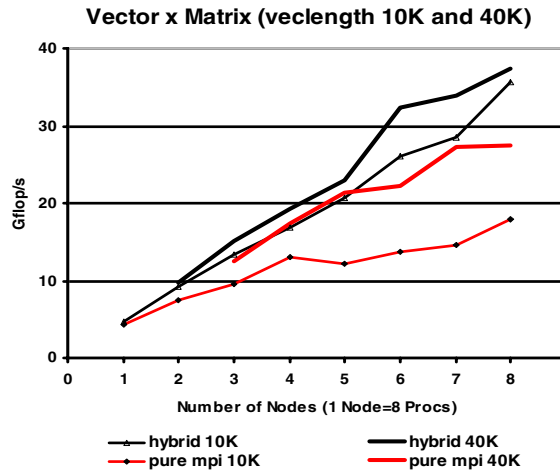


Fig. 7. Performance for vector by matrix multiplication.

optimally for a given cartesian processor grid. Given a block (IB, JB) of matrix C situated on a particular processor P, the required blocks of matrices A and B are pipelined through this processor in the course of the calculation. Between the communication steps, a normal DGEMM-call is performed. One can expect a nearly linear speed-up provided the block size is chosen large enough

to essentially circumvent MPI latency. For this test a fixed amount of memory per node was used; the matrix dimension correspondingly scales upward with the square root of the number of nodes used. Figure 8 shows the dependence of

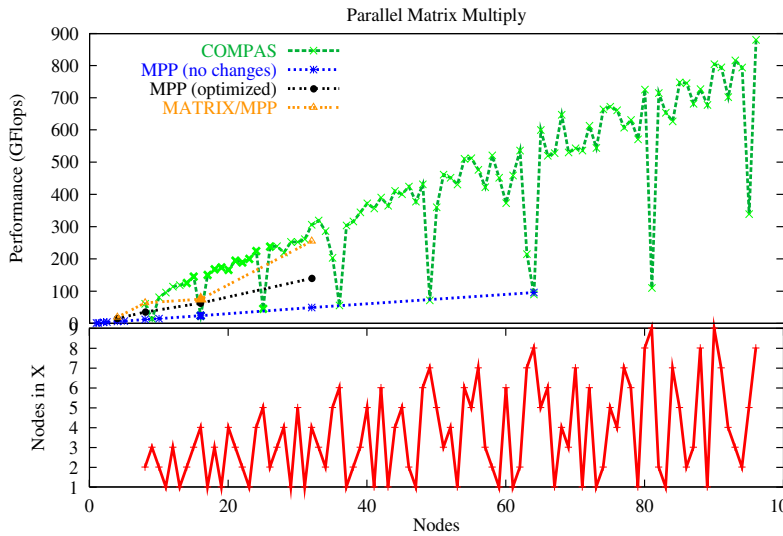


Fig. 8. Performance of parallel matrix multiplication.

performance on the number of nodes in various situations in the upper box, for the COMPAS case the lower box gives information about the node layout in the cartesian grid; e. g., for 21 nodes a 3x7 grid was used. In the COMPAS case, scaling is reasonably linear; for a yet unknown reason, square grids (4x4, 5x5, ...) appear to work particularly badly and should be avoided, at least with this particular communication pattern. Memory per node was 1.6 GByte. For the MPP (intra-node MPI) case, where 200 MByte of memory were used to obtain the same memory footprint per node as in the COMPAS case, runs with usually up to 256 processors (corresponding to 32 nodes) were performed. One obtains the very bad performance shown by the lowest curve if one simply recompiles the COMPAS code in scalar mode, using the scalar version of BLAS provided by Hitachi. The reason for this is simply that the latter library was not properly optimized at the time the tests were performed; as the "matmul_gemm" curve in Figure 9 shows, the code apparently did not optimally reuse the cache. Since the LRZ hand-coded routine ("matmul_opt2") works best for matrix sizes around 60-80, a smaller block size was chosen for a further run of the program yielding the second-lowest curve in Figure 10, which shows an improvement by a factor of 2.5. However, Hitachi also provides a proprietary library of highly optimized routines, MATRIX/MPP. Use of these – with a large block size – yields a performance comparable to COMPAS also for the internode case, as shown in the

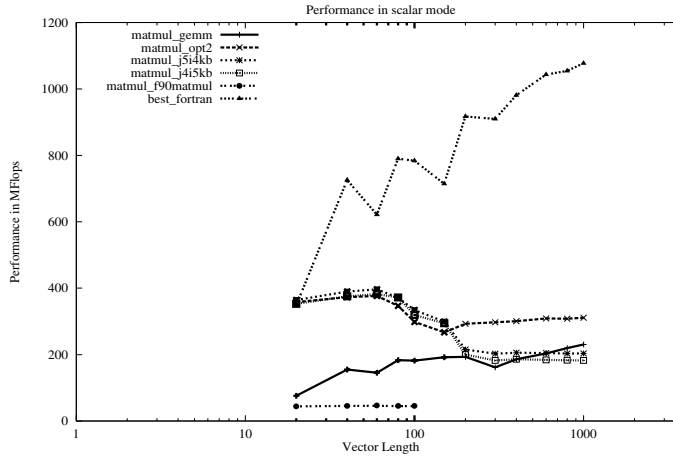


Fig. 9. Non-parallel (single-IP) matrix multiplication. The following variants are illustrated: 1. matmul_gemm: Hitachi BLAS, 2. matmul_opt2, matmul_j5i4kb, matmul_j4i5kb: blocking and loop unrolling done by hand, 3. matmul_f90matmul: Fortran 90 intrinsic, 4. best_fortran: Hitachi's MATRIX/MPP implementation.

third-lowest curve of Figure 8. The drawback of MATRIX/MPP is that there is a different API and presumably more working space has to be used.

4 Conclusion

Hitachi's SR8000-F1 installation easily manages to provide the computational power demanded by LRZ's requirements and Hitachi's own commitments. Our first tests indicate that generally more effective usage of the machine may be made by using the COMPAS mode as opposed to intra-node MPI (MPP-mode). The automatic parallelization features available with the Fortran and C compilers make it a relatively easy task to optimize high performance computing code.

5 Further Reading and Details

A Superscalar RISC Processor with 160 FPRs for Large Scale Scientific Processing:
`http://www.lrz-muenchen.de/services/compute/hlrb/system-en/Iccd99.pdf`
Node Architecture and Performance Evaluation of the Hitachi SR8000:
`http://www.lrz-muenchen.de/services/compute/hlrb/system-en/NodeArch.pdf`
An overview of the Hitachi SR8000-F1 by Hitachi may also be found at:
`http://www.hitachi-eu.com/hel/hpcc`
Some of the STREAM results were taken from:
`http://www.cs.virginia.edu/stream/top10/Bandwidth.html`

Index of Authors

Aberdeen, Douglas, 980
Acquaviva, Jean-Thomas, 539
Adle, Roxane, 340
Afrati, Foto, 288
Agha, Gul A., 1029
Agrawal, Divyakant, 427
Agrawal, Gagan, 625
Ahmed, Nawaaz, 368
Ahrem, Regine, 1315
Aiguier, Marc, 340
Al-Sadi, Jehad, 935
Alcover, Rosa, 909
Almeida, Francisco, 320
Antoniu, Gabriel, 1039
Arnau, Vicente, 1206
Arnold, Dorian C., 1213
Ast, Markus, 519
Atun, Murat, 234
Avresky, Dimiter, 1148

Babayan, Boris, 18
Bachmann, Dieter, 1213
Baden, Scott, 617
Bader, Michael, 795
Bader, Reinhold, 1351
Baiardi, Fabrizio, 218
Baker, Mark, 1115
Bal, Henri E., 690
Bała, Piotr, 511
Baldoni, Roberto, 609
Bampis, Evripidis, 288
Bandera, Gerardo, 331
Barrado, Cristina, 519
Barry, Ed, Jr., 739
Barton, John, 1031
Bartzis, Constantinos, 877
Baxter, Jonathan, 980
Bečka, Martin, 861
Beckert, Armin, 1315
Becuzzi, Primo, 218
Behrens, Jörn, 815
Benkner, Siegfried, 647
Benner, Peter, 824
Bermudo, Nerina, 194
Beyls, Kristof E., 998
Bianco, Mauro, 638
Bischof, Christian H., 86
Böhm, Klemens, 435
Bonhomme, Alice, 1110
Bouabdallah, Abdelmadjid, 600
Bougé, Luc, 1039
Bourgeois, Julien, 208
Brandes, Thomas, 647
Brehm, Matthias, 1351
Brezany, Peter, 1251
Brodlie, Ken, 1323
Bromling, Steven, 95
Bücker, H. Martin, 86
Budiu, Mihai, 969
Bungartz, Hans-Joachim, 771
Busch, Costas, 575
Butz, Torsten, 829
Buyya, Rajkumar, 1115

Cagnard, Paul-Jean, 767
Cai, Wentong, 189
Cain, Harold W., 108
Calder, Brad, 70
Cameron, Kirk W., 141
Campo, Renato, 849
Cappello, Peter, 1231
Caragiannis, Ioannis, 877
Cela, José, 519
Chang, Lung-Chung, 994
Chapman, Barbara, 329
Chassin de Kergommeaux, Jacques, 133
Chen, Ying, 1302
Chiao, Hsin-Ta, 1053
Chirivella, Vicente, 909
Chiti, Sarah, 218
Choudhary, Alok, 1263, 1292
Chung, Chung-Ping, 994
Clark, Terry W., 511
Coghlan, Brian A., 1143
Collard, Jean-François, 329
Corporaal, Henk, 349, 965, 1105
Costa, Vítor Santos, 744
Cotofana, Sorin, 965
Cunha, José C., 1313
Cunniffe, Ronan, 1143

Danelutto, Marco, 1175
Dang Tran, Frédéric, 1061
Daoudi, El Mostafa, 506
D'Apuzzo, Marco, 839
Darling, Gordon J., 500
Darlington, John, 686
Darte, Alain, 357, 405
David, Pierre, 1201
Davis, M. Kei, 739
Day, Khaled, 935
Decker, Thomas, 277
De Dios Hernández, Agustín, 550
Delaplace, Franck, 340
Demetriou, Neophytos, 575
D'Hollander, Erik H., 998
Díaz de Cerio, Luis, 591
Diderich, Claude, 405
Dobrev, Stefan, 927
Dongarra, Jack, 1213
Drozdowski, Maciej, 311
Duato, Jose, 875

Ebcioglu, Kemal, 939
Ebner, Ralf, 1351
Efraimidis, Pavlos S., 456
El Abbadi, Amr, 427
Espinosa, Antonio, 173

Faden, Michael, 1315
Fahringer, Thomas, 105
Farrens, Matthew K., 989, 1008
Feautrier, Paul, 1201
Fink, Torsten, 1223
Finta, Lucian, 288
Fischer, Rolf, 519
Fodor, Eugene F., 729
Ford, Rupert W., 395
Fox, Geoffrey, 1211
Franke, Hubertus, 242
Freytag, Johann-Christoph, 445

Gan, Boon-Ping, 189
Gao, Guang R., 625
Gemund, Arjan J.C. van, 272
Gengler, Marc, 405
Gentzsch, Wolfgang, 1313
Gerholt, Thomas, 1315
Gerndt, Michael, 45
Gerner, Philippe, 668
Gérodolle, Anne, 1061
Getov, Vladimir, 617
Giné, Francesc, 1165
Godlevsky, Alexander B., 754
Golby, David, 1345
Goldstein, Seth C., 969
González, Antonio, 194, 591
González, Daniel, 320
Gonzalez, Jesus A., 682
Gorlatch, Sergei, 617
Gottschling, Peter, 784
Grabs, Torsten, 435
Gray, Paul A., 709
Grelck, Clemens, 620
Grundmann, Tobias, 1081
Gubitoso, Marco Dimas, 160
Guirado, Fernando, 262
Gupta, Sandeep K.S., 600
Gurd, John R., 75
Gürsoy, Attila, 234

Hadj, Yahya Ould Mohamed El, 506
Hammond, Kevin, 739
Hanusse, Nicolas, 583
Harper, John S., 149
Hatcher, Philip J., 1039
Haveraaen, Magne, 758
Heber, Gerd, 625
Hedges, Richard, 1253
Heine, Felix, 415
Heinrich, Ralf, 1315
Heller, Helmut, 1351
Hempel, Rolf, 1251
Herlihy, Maurice, 575
Hernández, Porfidio, 1165
Hernández, Vicente, 824
Hirayama, Hiroyuki, 527
Hluchý, Ladislav, 754
Holliday, JoAnne, 427
Holmerin, Jonas, 762
Hong, Lee Wei, 700
Hoogerbrugge, Jan, 950
Hovland, Paul D., 86
Hrúz, Tomáš, 861
Hsu, Wen-Jing, 595
Hsu, Windsor W., 1302
Hu, Weiwu, 1132
Huedo, Eduardo, 199
Humes, Carlos, Jr., 160
Hyde, Daniel C., 1115

Igai, Mitsuyoshi, 527
Ijaha, Stephen E., 869
Ioki, Nobuhiro, 527
Ionescu, Felicia, 532
Ionescu, Mihail, 532
Ishihara, Takashi, 729
Ishikawa, Yutaka, 1071

Jalba, Andrei, 532
Jalby, William, 539
Jennings, Dave, 168
Ju, Jialin, 718
Juhasz, Zoltan, 1171

Kaklamanis, Christos, 835, 877
Kalantery, Nasser, 869
Kamachi, Tsunehiko, 1239
Kandemir, Mahmut T., 1263
Karkowski, Ireneusz, 349
Karl, Wolfgang, 851
Karypis, George, 252, 296
Kaur, Samian, 1332
Kelly, Paul H.J., 567, 617
Kerbyson, Darren J., 149
Kersken, Hans-Peter, 1315
Kesmarki, Laszlo, 1171
Kewley, John M., 65
Keyes, David E., 1
Khonsari, Ahmad, 900
Kielmann, Thilo, 690
Kik, Marcin, 471
Kikuchi, Sumio, 1023
Kindermann, Stephan, 1223
Klusik, Ulrike, 739
Knoop, Jens, 329
Koniges, Alice E., 1253, 1273
Konstantopoulos, Charalampos, 835
Kranakis, Evangelos, 583
Krizanc, Danny, 583
Kügeler, Edmund, 1315
Kumar, Rishi, 625
Kumar, Vipin, 252, 296
Kurmann, Christian, 1118
Kutyłowski, Mirosław, 455
Kyushima, Ichiro, 1023

Labarta, Jesús, 519
Laborda, Óscar, 519
Lafage, Thierry, 178
Laforenza, Domenico, 1211
Lancaster, David, 1340
Larriba-Pey, Josep-L., 940
von Laszewski, Gregor, 22
Leinberger, William, 252
Leon, Coromoto, 682
Li, Maozhen, 1345
Li, Tiejun, 729
Liao, Ching-Jung, 57
Lim, Chu-Cheow, 189
Lin, Hong, 481
Lindenmaier, Götz, 223
Lisper, Björn, 762
Liu, Haiming, 1132
Llabería, José María, 960
Llanos Ferraris, Diego R., 550
Llorente, Ignacio M., 199
Llosa, Josep, 194
Loidl, Hans-Wolfgang, 739
Low, Yoke-Hean, 189
Luque, Emilio, 173, 262, 1165
Lynch, Robert E., 481
Lysne, Olav, 890

MacBeth, Mark, 1039
MacDonald, Steve, 95
Manneback, Pierre, 506
Manniesing, Rashindra, 349
Manz, Hartmut, 519
Margalef, Tomas, 173
Marinescu, Dan C., 481
Marino, Marina, 839
Martonosi, Margaret, 1018
Massingill, Berna L., 678
Mateev, Nikolay, 379
Mattson, Timothy G., 678
Mavronicolas, Marios, 575
May, David, 545
Mayer auf der Heide, Friedhelm, 455
Mayer, Anthony, 686
Mayo, Rafael, 824
Mayr, Ernst W., 573
McGuigan, Keith, 1039
McKinley, Kathryn S., 223
Mechelli, Marco, 609
Mehra, Pankaj, 1148
Melas, Panagiotis, 183
Melideo, Giovanna, 609
Memik, Gokhan, 1263
Menon, Vijay, 379
Meuer, Hans Werner, 43

Meyer, Ulrich, 461
Meziane, Abdelouafi, 506
Midkiff, Samuel P., 329
Milenkovic, Aleksandar, 558
Milis, Ioannis, 288
Miller, Barton P., 45, 108
Milutinovic, Veljko, 558
Min, Geyong, 904
Mitschang, Bernhard, 425
Mittermaier, Christian, 1196
Mohr, Bernd, 123
Mohr, Marcus, 806
Monien, Burkhard, 277
Morancho, Enric, 960
More, Sachin, 1292
Moreira, Jose E., 242
Moreno, Luz Marina, 320
Moreno, Salvador, 1206
Mori, Paolo, 218
Motokawa, Keiko, 1023
Mueller, Frank, 1185
Mukherjee, Nandini, 75
Mulholland, Connor, 500
Muller, Henk, 545
Müller, Silvia, 537
Muralidhar, Rajeev, 1332
Muraoka, Yoichi, 22

Nagel, Wolfgang E., 105, 784
Namyst, Raymond, 1039
Naono, Ken, 527
Navarro, Carlos, 940
Neary, Michael O., 1231
Newhouse, Steven, 686
Ngo, Ton, 1031
Nieplocha, Jarek, 718
van Nieuwpoort, Rob V., 690
Nishiyama, Hiroyasu, 1023
Nolte, Jörg, 1071
Novillo, Diego, 389
Nudd, Graham R., 149

O'Boyle, Michael F.P., 395
Olivé, Àngel, 960
Olsson, Ronald A., 729
Orduña, Juan M., 1206
Ould-Khaoua, Mohamed, 900, 904, 935

Papaefstathiou, Efstathios, 149
Parashar, Manish, 1332
Pardalos, Panos M., 839
Pasquarelli, Antonello, 861
Pedroso, Hernâni, 1157
Peinl, Peter, 451
Peng, Shietung, 1086
Peyton Jones, Simon L., 739
Phipps, Alan, 1231
Piccoli, Fabiana, 682
Pingali, Keshav, 368, 379
Post, Peter, 1315
Preis, Robert, 277
Prieto, Manuel, 199
Printista, Marcela, 682
Priol, Thierry, 1239, 1313
Procter, Jim, 1323
Prodan, Radu, 65
Prost, Jean-Pierre, 1253
Prylli, Loïc, 1110
Pucci, Geppino, 638

Qiu, Ling, 595
Quintana-Ortí, Enrique S., 824

Rabenseifner, Rolf, 1273
Rădulescu, Andrei, 272
Ragde, Prabhakar, 455
Ramírez, Alex, 940
Ramirez, Rafael, 700
Rana, Omer F., 168, 1345
Rathmayer, Sabine, 47
Rauch, Felix, 1118
Raynal, Michel, 35, 605
Rebón Portillo, Álvaro J., 739
Reeve, Jeff S., 1340
Reinefeld, Alexander, 1211
Renault, Eric, 1201
René, Christophe, 1239
Resch, Michael, 479
Ricci, Laura, 218
Rich, Kevin D., 989, 1008
Richert, Thomas, 325
Richman, Steven, 1231
Riley, Graham D., 75
Ripoll, Ana, 262
Ritt, Marcus, 1081
Robles, Antonio, 882
Rocha, Ricardo, 744
Roda, José L., 682
Rodríguez, Casiano, 320
Rodrigues, Luís, 605

Rodriguez, Casiano, 682
Röhm, Uwe, 435
Roig, Concepció, 262
Romero, Luis F., 491
Romero, Sergio, 491
Rosenstiel, Wolfgang, 1081
Rüde, Ulrich, 771
Ruiz, Aurelio, 1206

Sahelices Fernández, Benjamín, 550
Sakr, Majd, 969
Salinger, Petr, 931
Sancho, José Carlos, 882
de Sande, Francisco, 682
Sanders, Beverly A., 678
Sanders, Peter, 461, 918
Santosa, Andrew E., 700
Sarkar, Prasenjit, 1284
Sato, Mitsuhisa, 1071
Schaeffer, Jonathan, 95, 389
Scheffner, Dieter, 445
Schek, Hans-Jörg, 435
Schimmler, Manfred, 1085
Schloegel, Kirk, 296
Schmidt, Bertil, 1095
Schnor, Bettina, 217
Scholz, Sven-Bodo, 620
Schot, Henjo, 1105
Schreiber, Andreas, 1315
Schreiner, Wolfgang, 1196
Schulz, Martin, 851
Schulz, Uwe, 519
Sedukhin, Stanislav, 1086
Seidel, Edward, 1211
Sen, Shondip, 545
Senar, Miquel A., 262
Seznec, André, 178
Sherwood, Timothy, 70
Shields, Matthew S., 1345
Shriver, Elizabeth, 1251
Shudo, Kazuyuki, 22
Shurbanov, Vladimir, 1148
Sibeyn, Jop F., 918
Silber, Georges-André, 357
Silva, Fernando, 744
Silva, João Gabriel, 1157
Sivasubramaniam, Anand, 242
Sloan, Terence M., 500
Slowik, Adrian, 415
Solsona, Francesc, 1165
Spaccamela, Alberto Marchetti, 609
Spies, François, 208
Spirakis, Paul G., 456
Srimani, Pradip K., 600
Stanca, Marian, 965
Stefanović, Darko, 1018
Stein, Benhur de Oliveira, 133
Stigliani, Massimiliano, 1175
Stillger, Michael, 445
Stöhr, Elena A., 395
Straatsma, Tjerk P., 718
Stenström, Per, 537
Stricker, Thomas M., 1118
Strietzel, Martin, 1315
von Stryk, Oskar, 829
Sun, Xian-He, 141
Sunderam, Vaidy S., 709
Svolos, Andreas, 835
Szafron, Duane, 95

Talbot, Sarah A.M., 567
Tavangarian, Djamshid, 1115
Temam, Olivier, 223
Thakur, Rajeev, 1251
Theiss, Ingebjørg, 890
Theobald, Kevin B., 625
Thulasiram, Ruppa K., 625
Tirado, Francisco, 199
Ton, Lee-Ren, 994
Toraldo, Gerardo, 839
Treumann, Richard, 1253
Trinder, Philip W., 739
Tvrdík, Pavel, 931

Unrau, Ronald C., 389

Valero, Mateo, 537, 940
Valero, Rodrigo, 1206
Valero-García, Miguel, 591
Vassiliadis, Stamatis, 537, 965
Vera, Xavier, 194
Vergados, Ioannis, 877
Violard, Eric, 668
Vivien, Frédéric, 405
Völk, Martin, 851
Vrt'o, Imrich, 927

Wagner, Claus, 1185
Walker, David W., 1313, 1345

Walker, Kip, 969
Wang, Hong, 984
Watson, William, 1148
White, Alison, 1253
Wilcox, Daniel V., 149
Winkler, Franz, 1196
Winter, Stephen C., 869
Wise, David S., 774
Wismüller, Roland, 849
Wolf, Felix, 123
Wolf, Klaus, 1315
Wolniewicz, Paweł, 311
Wolter, Thieß-Magnus, 829
Wood, Jason, 1323
Wright, Helen, 1323
Wu, Chi-Houng, 1053
Wylie, Brian J.N., 108

Yamamoto, Yusaku, 527
Yéh, Thomas Y., 984
Young, Honesty C., 1302
Yuan, Shyan-Ming, 1053

Zaluska, Ed J., 183
Zapata, Emilio L., 331, 491
Zavanella, Andrea, 658
Zenger, Christoph, 795
Zhang, Fuxin, 1132
Zhang, Yanyong, 242
Ziegler, Sibylle, 851
Zimmermann, Jens, 815