

Lecture Notes in Computer Science

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

2474

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Tokyo

Dieter Kranzlmüller Peter Kacsuk
Jack Dongarra Jens Volkert (Eds.)

Recent Advances in Parallel Virtual Machine and Message Passing Interface

9th European PVM/MPI Users' Group Meeting
Linz, Austria, September 29 – Oktober 2, 2002
Proceedings



Springer

Volume Editors

Dieter Kranzlmüller

Jens Volkert

Johannes Kepler Universität Linz

Abteilung für Graphische und Parallele Datenverarbeitung (GUP)

Institut für Technische Informatik und Telematik

Altenbergstr. 69, 4040 Linz, Austria

E-mail: {kranzlmueeller, volkert}@gup.jku.at

Peter Kacsuk

MTA SZTAKI, Hungarian Academy of Sciences

Computer and Automation Research Institute

Victor Hugo u. 18-22, 1132 Budapest, Hungary

E-mail: kacsuk@sztaki.hu

Jack Dongarra

University of Tennessee, Computer Science Department

Innovative Computing Laboratory

1122 Volunteer Blvd, Knoxville, TN 37996-3450, USA

E-mail: dongarra@cs.utk.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Recent advances in parallel virtual machine and message passing interface :
proceedings / 9th European PVM MPI Users' Group Meeting, Linz, Austria,
September 29 - Oktober 2, 2002 / Dieter Kranzlmüller ... (ed.). - Berlin ;
Heidelberg ; New York ; Hong Kong ; London ; Milan ; Paris ; Tokyo :
Springer, 2002

(Lecture notes in computer science ; Vol. 2474)

ISBN 3-540-44296-0

CR Subject Classification (1998): D.1.3, D.3.2, F.1.2, G.1.0, B.2.1, C.1.2

ISSN 0302-9743

ISBN 3-540-44296-0 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

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2002

Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein

Printed on acid-free paper SPIN: 10871356 06/3142 5 4 3 2 1 0

Preface

Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) are the most frequently used tools for programming according to the message passing paradigm, which is considered one of the best ways to develop parallel applications.

This volume comprises 50 selected contributions presented at the Ninth European PVM/MPI Users' Group Meeting, which was held in Linz, Austria, September 29 – October 2, 2002. The conference was organized by the Department for Graphics and Parallel Processing (GUP), Institute of Technical Computer Science and Telematics, Johannes Kepler University Linz, Austria/Europe.

This conference has been previously held on Santorini (Thera), Greece (2001), in Balatonfüred, Hungary (2000), Barcelona, Spain (1999), Liverpool, UK (1998), and Krakow, Poland (1997). The first three conferences were devoted to PVM and were held at the TU Munich, Germany (1996), ENS Lyon, France (1995), and the University of Rome, Italy (1994).

This conference has become a forum for users and developers of PVM, MPI, and other message passing environments. Interaction between those groups has proved to be very useful for developing new ideas in parallel computing and for applying some of those already existent to new practical fields. The main topics of the meeting were evaluation and performance of PVM and MPI, extensions and improvements to PVM and MPI, algorithms using the message passing paradigm, and applications in science and engineering based on message passing. In addition, this year's conference was extended to include PVM and MPI in metacomputing and grid environments in order to reflect the importance of this topic to the high performance computing community.

Besides the main track of contributed papers, the conference featured the two special sessions "CrossGrid – Development of Grid Environments for Interactive Applications" as well as "ParSim – Current Trends in Numerical Simulation for Parallel Engineering Environments". The conference also included two tutorials, one on "MPI on the Grid" and another one on "Parallel Application Development with the Hybrid MPI+OpenMP Programming Model", and invited talks on high performance numerical libraries, advances in MPI, process management, scalability and robustness in metacomputing, petascale computing, performance analysis in the grid, and making grid computing mainstream. These proceedings contain the papers of the 50 contributed presentations together with abstracts of the invited speakers' presentations.

The ninth Euro PVM/MPI conference was held together with DAPSYS 2002, the fourth Austrian-Hungarian Workshop on Distributed and Parallel Systems. Participants of the two events shared invited talks, tutorials, the vendors' session, and social events while contributed paper presentations proceeded in separate tracks in parallel. While Euro PVM/MPI is dedicated to the latest development of PVM and MPI, DAPSYS is a major event to discuss general aspects of dis-

tributed and parallel systems. In this way the two events were complementary to each other and participants of Euro PVM/MPI could benefit from the joint organization of the two events.

Invited speakers of the joint Euro PVM/MPI and DAPSYS conference were Marian Bubak, Jack Dongarra, Al Geist, Michael Gerndt, Bill Gropp, Zoltan Juhasz, Ewing Lusk, Bart Miller, Peter Slood, and Vaidy Sunderam. The tutorials were presented by Barbara Chapman as well as Bill Gropp and Ewing Lusk.

We would like to express our gratitude for the kind support of our sponsors (see below) and we thank the members of the Program Committee and the additional reviewers for their work in refereeing the submitted papers and ensuring the high quality of Euro PVM/MPI.

September 2002

Dieter Kranzlmüller
 Peter Kacsuk
 Jack Dongarra
 Jens Volkert

Organization

General Chair

Jack Dongarra (University of Tennessee, Knoxville, TN, USA)

Program Chairs

Peter Kacsuk (MTA SZTAKI, Budapest, Hungary)
Dieter Kranzlmüller (GUP, Joh. Kepler University Linz, Austria)
Jens Volkert (GUP, Joh. Kepler University Linz, Austria)

Local Organization

Department for Graphics and Parallel Processing (GUP),
Institute of Technical Computer Science and Telematics,
Johannes Kepler University Linz, Austria/Europe.

Program Committee

Vassil Alexandrov (University of Reading, United Kingdom)
Ranieri Baraglia (CNUCE Institute Italy)
Arndt Bode (Technical Univ. of Munich, Germany)
Marian Bubak (AGH, Cracow, Poland)
Jacques Chassin (LSR-IMAG, France)
de Kergommeaux (Univ. of Athens, Greece)
Yiannis Cotronis (New University of Lisbon, Portugal)
Frederic Desprez (INRIA, France)
Erik D'Hollander (University of Ghent, Belgium)
Jack Dongarra (University of Tennessee, Knoxville, USA)
Graham Fagg (HLRS, Germany)
Joao Gabriel (University of Coimbra, Portugal)
Al Geist (Oak Ridge National Laboratory, USA)
Michael Gerndt (Technical Univ. of Munich, Germany)
Andrzej Goscinski (Deakin University, Australia)
William Gropp (Argonne National Laboratory, USA)
Gundolf Haase (Joh. Kepler University Linz, Austria)
Rolf Hempel (DLR, Simulation Aerospace Center, Germany)
Ladislav Hluchy (Slovak Academy of Sciences, Slovakia)
Peter Kacsuk (MTA SZTAKI, Hungary)

VIII Organization

| | |
|----------------------|---|
| Dieter Kranzlmüller | (Joh. Kepler University Linz, Austria) |
| Jan Kwiatkowski | (Wroclaw University of Technology, Poland) |
| Domenico Laforenza | (CNUCE, Italy) |
| Laurent Lefevre | (INRIA / RESAM, France) |
| Thomas Ludwig | (University of Heidelberg, Germany) |
| Emilio Luque | (Autonomous University of Barcelona, Spain) |
| Ewing Lusk | (Argonne National Laboratory, USA) |
| Tomas Margalef | (Autonomous University of Barcelona, Spain) |
| Barton Miller | (University of Wisconsin, USA) |
| Shirley Moore | (University of Tennessee, USA) |
| Wolfgang Nagel | (Dresden University of Technology, Germany) |
| Benno J. Overeinder | (Vrije Universiteit Amsterdam, The Netherlands) |
| Rolf Rabenseifner | (University of Stuttgart, Germany) |
| Andrew Rau-Chaplin | (Dalhousie University, Canada) |
| Jeff Reeve | (University of Southampton, United Kingdom) |
| Ralf Reussner | (DSTC, Monash University, Australia) |
| Yves Robert | (ENS Lyon, France) |
| Casiano | |
| Rodriguez-Leon | (Universidad de La Laguna, Spain) |
| Wolfgang Schreiner | (Joh. Kepler University Linz, Austria) |
| Miquel Senar | (Autonomous University of Barcelona, Spain) |
| Vaidy Sunderam | (Emroy University, USA) |
| Francisco Tirado | (Universidad Complutense, Spain) |
| Bernard Tourancheau | (SUN labs Europe, France) |
| Jesper Larsson Träff | (NEC Europe Ltd, Germany) |
| Pavel Tvrdik | (Czech Technical University, Czech Republic) |
| Jens Volkert | (Joh. Kepler University Linz, Austria) |
| Jerzy Wasniewski | (Danish Computing Centre, Denmark) |
| Roland Wismüller | (Technical Univ. of Munich, Germany) |

Additional Reviewers

| | |
|---------------------|--------------------------|
| Panagiotis Adamidis | Francisco Almeida |
| Thomas Boenisch | Vitor Duarte |
| Jose Duato | James Kohl |
| Coromoto Leon | Elsa Macias |
| Paolo Palmerini | Raffaele Perego |
| Gavin Pringle | Enrique S. Quintana-Orti |
| Alvari Suarez | |

Sponsoring Institutions
(as of July 18, 2002)

IBM Deutschland GmbH
Myricom, Inc.
Microsoft Research
SGI (Silicon Graphics GmbH)
HP (Hewlett-Packard)
AMD (Advanced Micro Devices GmbH)
NEC Europe Ltd.
Dolphin Interconnect Solutions
SCALI AS
Platform Computing GmbH
Pallas GmbH
NAG (Numerical Algorithms Group Ltd.)
Johannes Kepler University Linz, Austria/Europe

Table of Contents

Invited Talks

| | |
|--|----|
| High Performance Computing, Computational Grid, and Numerical Libraries | 1 |
| <i>Jack Dongarra</i> | |
| Performance, Scalability, and Robustness in the Harness Metacomputing Framework | 3 |
| <i>Vaidy Sunderam</i> | |
| Surfing the Grid - Dynamic Task Migration in the Polder Metacomputer Project | 4 |
| <i>Dick van Albada and Peter Sloot</i> | |
| Petascale Virtual Machine: Computing on 100,000 Processors | 6 |
| <i>Al Geist</i> | |
| MPICH2: A New Start for MPI Implementations | 7 |
| <i>William Gropp</i> | |
| Making Grid Computing Mainstream | 8 |
| <i>Zoltan Juhasz</i> | |
| Process Management for Scalable Parallel Programs | 9 |
| <i>Ewing Lusk</i> | |
| A Security Attack and Defense in the Grid Environment | 10 |
| <i>Barton P. Miller</i> | |
| Performance Analysis: Necessity or Add-on in Grid Computing | 11 |
| <i>Michael Gerndt</i> | |

Tutorials

| | |
|--|----|
| MPI on the Grid | 12 |
| <i>William Gropp and Ewing Lusk</i> | |
| Parallel Application Development with the Hybrid MPI+OpenMP Programming Model | 13 |
| <i>Barbara Chapman</i> | |

Special Session: CrossGrid

| | |
|---|----|
| CrossGrid and Its Relatives in Europe | 14 |
| <i>Marian Bubak and Michał Turala</i> | |

| | |
|---|----|
| Towards the CrossGrid Architecture | 16 |
| <i>Marian Bubak, Maciej Malawski, and Katarzyna Zajac</i> | |
| Application of Component-Expert Technology for Selection of Data-Handlers in CrossGrid | 25 |
| <i>Lukasz Dutka and Jacek Kitowski</i> | |
| Training of Neural Networks: Interactive Possibilities in a Distributed Framework | 33 |
| <i>O. Ponce, J. Cuevas, A. Fuentes, J. Marco, R. Marco, C. Martínez-Rivero, R. Menéndez, and D. Rodríguez</i> | |
| An Infrastructure for Grid Application Monitoring | 41 |
| <i>Bartosz Baliś, Marian Bubak, Włodzimierz Funika, Tomasz Szepieniec, and Roland Wismüller</i> | |
| The CrossGrid Performance Analysis Tool for Interactive Grid Applications | 50 |
| <i>Marian Bubak, Włodzimierz Funika, and Roland Wismüller</i> | |

Special Session: ParSim

| | |
|---|----|
| Current Trends in Numerical Simulation for Parallel Engineering Environments | 61 |
| <i>Carsten Trinitis and Martin Schulz</i> | |
| Automatic Runtime Load Balancing of Dedicated Applications in Heterogeneous Environments | 62 |
| <i>Siegfried Höfinger</i> | |
| A Contribution to Industrial Grid Computing | 70 |
| <i>Andreas Blaszczyk and Axel Uhl</i> | |
| Parallel Computing for the Simulation of 3D Free Surface Flows in Environmental Applications | 78 |
| <i>Paola Causin and Edie Miglio</i> | |
| Testbed for Adaptive Numerical Simulations in Heterogeneous Environments | 88 |
| <i>Tiberiu Rotaru and Hans-Heinrich Nägeli</i> | |
| Simulating Cloth Free-Form Deformation with a Beowulf Cluster | 96 |
| <i>Conceição Freitas, Luís Dias, and Miguel Dias</i> | |

Applications Using MPI and PVM

| | |
|---|-----|
| Concept of a Problem Solving Environment for Flood Forecasting | 105 |
| <i>Ladislav Hluchy, Viet Dinh Tran, Ondrej Habala, Jan Astalos, Branislav Simo, and David Froehlich</i> | |

| | |
|---|-----|
| A Comprehensive Electric Field Simulation Environment on Top of SCI ... | 114 |
| <i>Carsten Trinitis, Martin Schulz, and Wolfgang Karl</i> | |
| Application of a Parallel Virtual Machine for the Analysis of a Luminous Field | 122 |
| <i>Leszek Kasprzyk, Ryszard Nawrowski, and Andrzej Tomczewski</i> | |
| Solving Engineering Applications with LAMGAC over MPI-2 | 130 |
| <i>Elsa M. Macías and Alvaro Suárez</i> | |
| Distributed Image Segmentation System by a Multi-agents Approach (Under PVM Environment) | 138 |
| <i>Yacine Kabir and A. Belhadj-Aissa</i> | |

Parallel Algorithms Using Message Passing

| | |
|--|-----|
| Parallel Global Optimization of High-Dimensional Problems | 148 |
| <i>Siegfried Höfinger, Torsten Schindler, and András Aszódi</i> | |
| Adjusting the Lengths of Time Slices when Scheduling PVM Jobs with High Memory Requirements | 156 |
| <i>Francesc Giné, Francesc Solsona, Porfidio Hernández, and Emilio Luque</i> | |
| A PVM-Based Parallel Implementation of the REYES Image Rendering Architecture | 165 |
| <i>Oscar Lazzarino, Andrea Sanna, Claudio Zunino, and Fabrizio Lamberti</i> | |
| Enhanced File Interoperability with Parallel MPI File-I/O in Image Processing | 174 |
| <i>Douglas Antony Louis Piriya Kumar, Paul Levi, and Rolf Rabenseifner</i> | |
| Granularity Levels in Parallel Block-Matching Motion Compensation | 183 |
| <i>Florian Tischler and Andreas Uhl</i> | |
| An Analytical Model of Scheduling for Conservative Parallel Simulation ... | 191 |
| <i>Ha Yoon Song, Junghwan Kim, and Kyun Rak Chong</i> | |
| Parallel Computation of Pseudospectra Using Transfer Functions on a MATLAB-MPI Cluster Platform | 199 |
| <i>Constantine Bekas, Efrosini Kokipoulou, Efstratios Gallopoulos, and Valeria Simoncini</i> | |
| Development and Tuning of Irregular Divide-and-Conquer Applications in DAMPVM/DAC | 208 |
| <i>Pawel Czarnul</i> | |
| Observations on Parallel Computation of Transitive and Max-Closure Problems | 217 |
| <i>Aris Pagourtzis, Igor Potapov, and Wojciech Rytter</i> | |

| | |
|---|-----|
| Evaluation of a Nearest-Neighbor Load Balancing Strategy for Parallel Molecular Simulations in MPI Environment | 226 |
| <i>Angela Di Serio and María B. Ibáñez</i> | |

Programming Tools for MPI and PVM

| | |
|--|-----|
| Application Recovery in Parallel Programming Environment | 234 |
| <i>Giang Thu Nguyen, Viet Dinh Tran, and Margareta Kotocova</i> | |
| IP- OORT : A Parallel Remeshing Toolkit | 243 |
| <i>Éric Malouin, Julien Dompierre, François Guibault, and Robert Roy</i> | |
| Modular MPI and PVM Components | 252 |
| <i>Yiannis Cotronis and Zacharias Tsiatsoulis</i> | |
| Communication Infrastructure in High-Performance Component-Based Scientific Computing | 260 |
| <i>David E. Bernholdt, Wael R. Elwasif, and James A. Kohl</i> | |
| On Benchmarking Collective MPI Operations | 271 |
| <i>Thomas Worsch, Ralf Reussner, and Werner Augustin</i> | |

Implementations of MPI and PVM

| | |
|---|-----|
| Building Library Components that Can Use Any MPI Implementation | 280 |
| <i>William Gropp</i> | |
| Stampi-I/O: A Flexible Parallel-I/O Library for Heterogeneous Computing Environment | 288 |
| <i>Yuichi Tsujita, Toshiyuki Imamura, Hiroshi Takemiya, and Nobuhiro Yamagishi</i> | |
| (Quasi-) Thread-Safe PVM and (Quasi-) Thread-Safe MPI without Active Polling | 296 |
| <i>Tomas Plachetka</i> | |
| An Implementation of MPI-IO on Expand: A Parallel File System Based on NFS Servers | 306 |
| <i>Alejandro Calderón, Félix García, Jesús Carretero, Jose M. Pérez, and Javier Fernández</i> | |
| Design of DMPI on DAWNING-3000 | 314 |
| <i>Wei Huang, Zhe Wang, and Jie Ma</i> | |
| MPICH-CM: A Communication Library Design for a P2P MPI Implementation | 323 |
| <i>Anton Selikhov, George Bosilca, Cecile Germain, Gilles Fedak, and Franck Cappello</i> | |

| | |
|---|-----|
| Design and Implementation of MPI on Portals 3.0 | 331 |
| <i>Ron Brightwell, Arthur B. Maccabe, and Rolf Riesen</i> | |
| Porting PVM to the VIA Architecture Using a Fast Communication Library | 341 |
| <i>Roberto Espenica and Pedro Medeiros</i> | |
| LICO: A Multi-platform Channel-Based Communication Library | 349 |
| <i>Moreno Coli, Paolo Palazzari, and Rodolfo Rughi</i> | |
| Notes on Nondeterminism in Message Passing Programs | 357 |
| <i>Dieter Kranzlmüller and Martin Schulz</i> | |

Extensions of MPI and PVM

| | |
|--|-----|
| Web Remote Services Oriented Architecture for Cluster Management | 368 |
| <i>Josep Jorba, Rafael Bustos, Ángel Casquero, Tomàs Margalef, and Emilio Luque</i> | |
| Improving Flexibility and Performance of PVM Applications by Distributed Partial Evaluation | 376 |
| <i>Bartosz Krysztop and Henryk Krawczyk</i> | |
| Ready-Mode Receive: An Optimized Receive Function for MPI | 385 |
| <i>Ron Brightwell</i> | |
| Improved MPI All-to-all Communication on a Giganet SMP Cluster | 392 |
| <i>Jesper Larsson Träff</i> | |
| Fujitsu MPI-2: Fast Locally, Reaching Globally | 401 |
| <i>Georg Bißeling, Hans-Christian Hoppe, Alexander Supalov, Pierre Lagier, and Jean Latour</i> | |

Performance Analysis and Optimization

| | |
|---|-----|
| Communication and Optimization Aspects on Hybrid Architectures | 410 |
| <i>Rolf Rabenseifner</i> | |
| Performance Analysis for MPI Applications with SCALEA | 421 |
| <i>Hong-Linh Truong, Thomas Fahringer, Michael Geissler, and Georg Madsen</i> | |
| A Performance Study of Load Balancing Strategies for Approximate String Matching on an MPI Heterogeneous System Environment | 432 |
| <i>Panagiotis D. Michailidis and Konstantinos G. Margaritis</i> | |
| An Analytical Model for Pipeline Algorithms on Heterogeneous Clusters | 441 |
| <i>F. Almeida, D. González, L.M. Moreno, and C. Rodríguez</i> | |

| | |
|--|-----|
| Architectures for an Efficient Application Execution in a Collection of HNOWS | 450 |
| <i>A. Furtado, A. Rebouças, J.R. de Souza, D. Rexachs, and E. Luque</i> | |
| Author Index | 461 |

High Performance Computing, Computational Grid, and Numerical Libraries

Jack Dongarra

Innovative Computing Laboratory (ICL)

University of Tennessee

Knoxville, TN

dongarra@cs.utk.edu

<http://www.netlib.org/utk/people/JackDongarra>

Abstract. In this talk we will look at how High Performance computing has changed over the last 10-year and look toward the future in terms of trends. In addition, we advocate the ‘Computational Grids’ to support ‘large-scale’ applications. These must provide transparent access to the complex mix of resources - computational, networking, and storage - that can be provided through aggregation of resources. We will look at how numerical library software can be run in an adaptive fashion to take advantage of available resources.

In the last 50 years, the field of scientific computing has seen rapid, sweeping changes—in vendors, in architectures, in technologies, and in the users and uses of high-performance computer systems. The evolution of performance over the same 50-year period, however, seems to have been a very steady and continuous process. Moore’s law is often cited in this context, and, in fact, a plot of the peak performance of the various computers that could be considered the “supercomputers” of their times clearly shows that this law has held for almost the entire lifespan of modern computing. On average, performance has increased every decade by about two orders of magnitude.

Two statements have been consistently true in the realm of computer science: (1) the need for computational power is always greater than what is available at any given point, and (2) to access our resources, we always want the simplest, yet the most complete and easy to use interface possible. With these conditions in mind, researchers have directed considerable attention in recent years to the area of grid computing. The ultimate goal is the ability to plug any and all of our resources into a computational grid and draw from these resources—this is analogous to the electrical power grid, much as we plug our appliances into electrical sockets today.

Advances in networking technologies will soon make it possible to use the global information infrastructure in a qualitatively different way—as a computational as well as an information resource. As described in the recent book “The Grid: Blueprint for a New Computing Infrastructure,” this “Grid” will connect the nation’s computers, databases, instruments, and people in a seamless web of computing and distributed intelligence, that can be used in an on-demand fashion as a problem-solving resource

in many fields of human endeavor—and, in particular, for science and engineering.

The availability of Grid resources will give rise to dramatically new classes of applications, in which computing resources are no longer localized, but distributed, heterogeneous, and dynamic; computation is increasingly sophisticated and multidisciplinary; and computation is integrated into our daily lives, and hence subject to stricter time constraints than at present. The impact of these new applications will be pervasive, ranging from new systems for scientific inquiry, through computing support for crisis management, to the use of ambient computing to enhance personal mobile computing environments.

In this talk we will explore the issues of developing a prototype system designed specifically for the use of numerical libraries in the grid setting.

Performance, Scalability, and Robustness in the Harness Metacomputing Framework

Vaidy Sunderam

Emory University
Atlanta, GA 30322, USA

vss@emory.edu

<http://mathcs.emory.edu/~vss>

Abstract. Harness is a software framework and methodology that facilitates distributed metacomputing via a reconfigurable plugin-based architecture. The current version of Harness emulates multiple concurrent programming environments including PVM, MPI, JavaSpaces, and generic task parallelism models. Supporting multiple parallel programming models within a unified framework has high utility and flexibility, and is capable of evolving as needed. Harness, a modular software system, is able to deliver such functionality and support legacy as well as new programming paradigms without loss of efficiency. In a set of benchmarking experiments, the performance of applications executed using the Harness-PVM system was within a few percent of their performance using native PVM. Furthermore, the Harness-PVM system scales well and exhibits increased tolerance to failures, due to its event-based distributed architecture. Details of the Harness infrastructure, features that contributed to enhanced scalability and failure-resilience, and performance results are presented.

Surfing the Grid - Dynamic Task Migration in the Polder Metacomputer Project

Dick van Albada and Peter Sloot

Section Computational Science, Universiteit van Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
`dick@science.uva.nl`

Abstract. Traditionally, PVM and MPI programs live on message passing systems, from clusters of non-dedicated workstations to MPP machines. The performance of a parallel program in such an environment is usually determined by the single least performing task in that program. In a homogeneous, stable environment, such as an MPP machine, this can only be repaired by improving the workload balance between the individual tasks. In a cluster of workstations, differences in the performance of individual nodes and network components can be an important cause of imbalance. Moreover, these differences will be time dependent as the load generated by other users plays an important role. Worse yet, nodes may be dynamically removed from the available pool of workstations. In such a dynamically changing environment, redistributing tasks over the available nodes can help to maintain the performance of individual programs and of the pool as a whole. Condor [1] solves this task migration problem for sequential programs. However, the migration of tasks in a parallel program presents a number of additional challenges, for the migrator as well as for the scheduler. For PVM programs, there are a number of solutions, including Dynamite [2]; Hector [3] was designed to migrate MPI tasks and to checkpoint complete MPI programs. The latter capability is very desirable for long-running programs in an unreliable environment.

This brings us to the Grid, where both performance and availability of resources vary dynamically and where reliability is an important issue. Once again, Livny with his Condor-G [4] provides a solution for sequential programs, including provisions for fault-tolerance. In the Polder Metacomputer Project, based on our experience with Dynamite, we are currently investigating the additional challenges in creating a task-migration and checkpointing capability for the Grid environment. This includes the handling of shared resources, such as open files; differences in administrative domains, etc. Eventually, the migration of parallel programs will allow large parallel applications to surf the Grid and ride the waves in this highly dynamic environment.

References

- [1] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, Checkpoint and migration of Unix processes in the Condor distributed processing system, Technical Report 1346, University of Wisconsin, WI, USA, 1997. 4

- [2] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, G. D. van Albada, B. J. Overeinder, P. M. A. Sloot, The implementation of Dynamite – an environment for migrating PVM tasks, *Operating Systems Review*, vol. 34, nr 3 pp. 40–55. Association for Computing Machinery, Special Interest Group on Operating Systems, July 2000. 4
- [3] J. Robinson, S. H. Russ, B. Flachs, B. Heckel, A task migration implementation of the Message Passing Interface, *Proceedings of the 5th IEEE international symposium on high performance distributed computing*, 61–68, 1996. 4
- [4] J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke, Condor-G: A Computation Management Agent for Multi-Institutional Grids, *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)* San Francisco, California, August 7–9, 2001. 4

Petascale Virtual Machine: Computing on 100,000 Processors

Al Geist

Oak Ridge National Laboratory
PO Box 2008, Oak Ridge, TN 37831-6367
gst@ornl.gov
<http://www.csm.ornl.gov/~geist>

Abstract. In the 1990s the largest machines had a few thousand processors and PVM and MPI were key tools to making these machines useable. Now with the growing interest in Internet computing and the design of cellular architectures such as IBM's Blue Gene computer, the scale of parallel computing has suddenly jumped to 100,000 processors or more. This talk will describe recent work at Oak Ridge National Lab on developing algorithms for petascale virtual machines and the development of a simulator, which runs on a Linux cluster, that has been used to test these algorithms on simulated 100,000 processor systems. This talk will also look at the Harness software environment and how it may be useful to increase the scalability, fault tolerance, and adaptability of applications on large-scale systems.

MPICH2: A New Start for MPI Implementations

William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL
`gropp@mcs.anl.gov`
<http://www.mcs.anl.gov/~gropp>

Abstract. This talk will describe MPICH2, an all-new implementation of MPI designed to support both MPI-1 and MPI-2 and to enable further research into MPI implementation technology. To achieve high-performance and scalability and to encourage experimentation, the design of MPICH2 is strongly modular. For example, the MPI topology routines can easily be replaced by implementations tuned to a specific environment, such as a geographically dispersed computational grid. The interface to the communication layers has been designed to exploit modern interconnects that are capable of remote memory access but can be implemented on older networks. An initial, TCP-based implementation, will be described in detail, illustrating the use of a simple, communication-channel interface. A multi-method device that provides TCP, VIA, and shared memory communication will also be discussed. Performance results for point-to-point and collective communication will be presented. These illustrate the advantages of the new design: the point-to-point TCP performance is close to the raw achievable latency and bandwidth, and the collective routines are significantly faster than the “classic” MPICH versions (more than a factor of two in some cases). Performance issues that arise in supporting `MPI_THREAD_MULTIPLE` will be discussed, and the role of a proper choice of implementation abstraction in achieving low-overhead will be illustrated with results from the MPICH2 implementation.

Scalability to tens or hundreds of thousands of processors is another goal of the MPICH2 design. This talk will describe some of the features of MPICH2 that address scalability issues and current research targeting a system with 64K processing elements.

Making Grid Computing Mainstream

Zoltan Juhasz

Department of Information Systems, University of Veszprem
Veszprem, Egyetem u. 10., H-8200 Hungary
juhasz@irt.vein.hu
<http://www.irt.vein.hu/~juhasz>

Abstract. The past few years have been good for grid computing. Interest in grid computing has increased exceptionally, numerous academic and industrial research groups are now working on grid projects investigating theoretical and technology-related problems, or developing scientific or commercial grid applications – grid computing even reached the world of politics, it is on the agenda of several governments. The results of these research efforts have shown the large potential of grid systems. It has also become evident that grids should not be limited to only computational problems, this technology could be beneficial in many other areas of our life. Consequently, grid computing technology is now moving from computational grids to service-oriented grids. This new type of grid – grid of services – could provide pervasive access to scientific, commercial, etc. services to anyone, any time, from anywhere and any device.

The talk will look at the recent developments in service-oriented grids, discuss the challenges we face in creating such very large scale systems, and concentrate on architectural, middleware and programming issues. I will also discuss the expected impact of mobile devices, and show how the Java platform, especially Jini technology can contribute to the creation of reliable service grids, and create an unparalleled opportunity to bring grid computing into the mainstream and make it available to the widest developer and user community.

Process Management for Scalable Parallel Programs

Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL
`lusk@mcs.anl.gov`
<http://www.mcs.anl.gov/~lusk>

Abstract. Large-scale parallel programs present multiple problems in process management, from scalable process startup to runtime monitoring and signal delivery, to rundown and cleanup. Interactive parallel jobs present special problems in management of standard I/O.

In this talk we will present an approach that addresses these issues. The key concept is that of a process management interface, which is used by application layers such as MPI implementations or the runtime systems for languages like UPC or Co-Array Fortran, and implemented by vendor-supplied or publicly available process management systems.

We will describe multiple implementations of such a process management interface, focusing on MPD, which is distributed with the MPICH implementation of MPI but is independent of it. MPD provides process management support for the MPICH-2 implementation of MPI, described elsewhere in this conference, as well as scalable process startup of large interactive jobs. It cooperates with various types of parallel tools such as monitors and debuggers.

Finally, we will describe how this or any process management systems can be integrated into a complete scalable systems software solution, using the interfaces under development by a broadly based group attempting to define a cluster system software architecture.

A Security Attack and Defense in the Grid Environment

Barton P. Miller

Computer Science Department
University of Wisconsin, Madison, WI
`bart@cs.wisc.edu`
<http://www.cs.wisc.edu/~bart>

Abstract. Programs in execution have long been considered to be immutable objects. Object code and libraries are emitted by the compiler, linked and then executed; any changes to the program require revisiting the compile or link steps. In contrast, we consider a running program to be an object that can be examined, instrumented, and re-arranged on the fly. The DynInst API provides a portable library for tool builders to construct tools that operate on a running program. Where previous tools might have required a special compiler, linker, or run-time library, tools based on DynInst can operate directly on unmodified binary programs during execution. I will discuss how this technology can be used to subvert system security and present an interesting scenario for security vulnerability in Grid computing. The example comes from an attack that we made on the Condor distributed scheduling system.

For this attack, we created "lurker" processes that can be left latent on a host in the Condor pool. These lurker processes lie in wait for subsequent Condor jobs to arrive on the infected host. The lurker then uses Dyninst to attach to the newly-arrived victim job and take control. Once in control, the lurker can cause the victim job to make requests back to its home host, causing it execute almost any system call it would like.

Using techniques similar to those in intrusion detection, I show how to automatically construct a nondeterministic finite automata from the binary code of the Condor job, and use this NFA while the job is executing to check that it is not acting out of character.

Performance Analysis: Necessity or Add-on in Grid Computing

Michael Gerndt

Technische Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-85748 Garching, Germany
gerndt@in.tum.de
<http://www.in.tum.de/~gerndt>

Abstract. Performance analysis tools are a necessity in parallel computing for creating highly tuned applications. Very powerful techniques enabling profiling tools, trace visualization tools, and automatic analysis tools have been developed to assist the programmer.

This presentation will discuss the requirements and the scenarios for performance analysis in Grid Computing. It will cover the current designs and the tools being under development in various grid projects. It will also introduce the grid-related work of the APART working group (www.fz-juelich.de/apart) focusing on automatic performance analysis.

MPI on the Grid

William Gropp and Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL

`gropp@mcs.anl.gov`

`http://www.mcs.anl.gov/~gropp`

`lusk@mcs.anl.gov`

`http://www.mcs.anl.gov/~lusk`

Abstract. This tutorial will cover parallel programming with the MPI message passing interface, with special attention paid to the issues that arise in a computational grid environment. After a summary of MPI programming, we will address the issue of process management, first in a single administrative domain and then across multiple administrative domains. We will describe the use of the MPD process management system with MPI implementations that support the Process Management Interface, and the MPICH-G2 MPI implementation for Globus-based distributed environments.

We will discuss grid-specific application programming issues such as latency-reduction techniques using message aggregation and the use of topology-aware collective operations. Remote I/O is one of the best aspects of a grid environment, and so the use of MPI-2 I/O operations will be covered. We will conclude with a discussion of fault-tolerance issues in MPI programs, including programming techniques and the state of current implementations.

Parallel Application Development with the Hybrid MPI+OpenMP Programming Model

Barbara Chapman

Dept. of Computer Science, University of Houston

Houston, Texas, USA

chapman@cs.uh.edu

<http://www.cs.uh.edu/~chapman>

Abstract. Many parallel platforms in use today are clusters of SMP systems, connected by Ethernet or one of the high-speed networks available. Some provide global memory addressing. Codes developed to run on these machines are often written using the MPI library for exchanging data. Increasingly, however, application developers have begun exploring the use of OpenMP in conjunction with MPI as a programming model. OpenMP is an industry standard for shared memory parallel program. Unlike MPI, it is based upon a set of directives that are inserted into a Fortran or C/C++ code and translated by a compiler into an explicitly parallel code. When used together with MPI, OpenMP is normally used to exploit the shared memory within each of the SMPs in the target platform.

In this tutorial, we review the OpenMP API for shared memory parallel programming, and then consider how OpenMP directives may be inserted into an existing MPI program. In order to understand how to do this well, we also look at performance issues related to the use of OpenMP. We complete the tutorial by discussing the potential benefits of this model and reporting on experiences already gained in a variety of efforts that have developed codes under this hybrid model.

CrossGrid and Its Relatives in Europe^{*}

Marian Bubak^{1,2} and Michał Turała^{3,2}

¹ Institute of Computer Science, AGH, al.
Mickiewicza 30, 30-059 Kraków, Poland

² Academic Computer Centre – CYFRONET
Nawojki 11, 30-950 Kraków, Poland

³ Institute of Nuclear Physics, High Energy
Physics Department, Kraków, Poland
bubak@uci.agh.edu.pl
michal.turala@cern.ch

The first part of this lecture presents the current status of the EU CrossGrid Project which develops grid services and programming tools for interactive applications (see [1] and six papers in this Proceedings volume).

The second part analyses the results of EU IST-funded research activity in this field: new grid-enabled applications, extension of grid technologies with new services and tools, and experience with testbed deployment. DataGrid develops middleware and testbeds for high energy physics, Earth observation and biology research [2]. The objective of the DATATAG project is to construct a transatlantic grid testbed [3]. GRIDLAB focuses on a toolkit for large scale simulations on the grid [4]. DAMIEN extends the MPI to grid computing [5], EUROGRID develops middleware for seamless access to HPC centers [6], while GRIP addresses interoperability between Unicore and Globus [7]. AVO will establish a virtual observatory for astronomy [8], and EGSO is working on a solar observatory virtual archive. GRIA is focused on e-business applications [9]. These projects collaborate within the framework of the GRIDSTART [10]. The family of EU grid projects is still growing and its new members [11] will address the following topics: on-demand flow simulation (FLOWGRID), grid computing for molecular science and engineering (OpenMolGrid), grid search and categorization (GRACE), corporate ontology (COG), environment for semantic web (MOSES), bio-technology information and knowledge (BioGrid), medical simulation services (GEMSS), self e-learning networks (SeLeNe), and a federated mammogram database on a grid (MammoGrid). This will result in a solid basis for the IST-FP6 program where grids will be applied to solve complex problems.

References

- [1] <http://www.eu-crossgrid.org> 14
- [2] <http://www.eu-datagrid.org> 14
- [3] <http://www.datatag.org/> 14
- [4] <http://www.gridlab.org/> 14
- [5] <http://www.hlr.de/organization/pds/projects/damien/> 14

^{*} Funded by the European Commission, project IST-2001-32243, CrossGrid

- [6] <http://www.eurogrid.org/> 14
- [7] <http://www.grid-interoperability.org/> 14
- [8] <http://www.eso.org/avo/> 14
- [9] <http://www.it-innovation.soton.ac.uk/research/grid/gria.shtml> 14
- [10] <http://www.gridstart.org/> 14
- [11] <http://www.cordis.lu/ist> 14

Towards the CrossGrid Architecture^{*}

Marian Bubak^{1,2}, Maciej Malawski¹, and Katarzyna Zajac¹

¹ Institute of Computer Science, AGH, al.
Mickiewicza 30, 30-059 Kraków, Poland

² Academic Computer Centre – CYFRONET
Nawojki 11, 30-950 Kraków, Poland
{bubak,malawski,kzajac}@uci.agh.edu.pl

phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54

Abstract. In this paper, basing on the software requirements specifications, we present the CrossGrid architecture. The architecture definition consists of descriptions of functionality of new tools and grid services and indicates where interfaces should be defined. It ought to be perceived as a starting point for a more detailed description of interfaces and technology used for their implementation. The components of the CrossGrid architecture are modular and they are organized in the following layers: applications, supporting tools, application development support, specific grid services, generic grid services, and a fabric layer.

Keywords: Grid, architecture, services, interactive applications

1 Introduction

The applications of the CrossGrid Project [6, 4] are characterized by interaction with a person in a processing loop. Each application requires a response from the Grid to an action by that person in different time scales: from real time through intermediate delays to long waiting periods. The applications are simultaneously compute- and data-intensive. The following applications are currently developed:

- simulation and visualization for surgical procedures,
- flooding crisis team decision support,
- distributed data analysis in high-energy physics,
- air pollution combined with weather forecasting.

Development of these applications for the Grid requires *new tools* for verification of parallel source code, performance prediction, performance evaluation and monitoring. Next, both applications and tools need *new Grid services* for application monitoring, efficient distributed data access, and specific resource management. To run the applications on the Grid in an easy and transparent way, without going into details of the Grid structure and operation users require user-friendly portals and roaming personalized portals.

^{*} Partly funded by 7th European Commission, project IST-2001-32243, CrossGrid

The CrossGrid development exploits the available achievements of Globus [12], DataGrid [7] and other related projects in a way which enables their interoperability [8]. The elaborated methodology, generic application architecture, programming environment, and new Grid services will be validated and tested thoroughly on the testbeds [6].

2 CrossGrid Applications

2.1 Simulation and Visualization for Surgical Procedures

The application will be a distributed near real-time simulation with a user interacting in Virtual Reality and other interactive display environments [6]. A 3-D model of arteries will be the input to a simulation for blood flow and the results will be presented in a VR environment. The user will change the structure of the arteries, thus mimicking an interventional or surgical procedure. The effects will be analysed in near real-time and the results will be presented to the user in VR. The software will consist of several components, which may be executed at different locations on different hardware platforms.

2.2 Flooding Crisis Team Decision Support

This application will be a support system for a virtual organization for flood forecasting [6]. Flood forecasting requires meteorological simulations of different resolutions, from mesoscale to storm-scale. Next, the hydrological models are used to determine the discharge from the affected area, and with this information hydraulic models simulate flow through various river structures to predict the impact of the flood. The models used in flood forecasting require a large amount of temporary storage as they produce massive output sets. A visualization tool is required to display meteorological forecasts, hydrological status of areas, and estimated water flows. The results of simulations can be interactively reviewed, some of them accepted and forwarded to the next step, some rejected or re-run with modified inputs.

2.3 Distributed Data Analysis in High-Energy Physics

This application will address access to large distributed databases in the Grid environment and development of distributed data-mining techniques suited to the high-energy physics field [6]. For this interactive analysis location of data plays a central role as it will run on O(TB) of distributed data, so transfer/replication times for the whole dataset will be in range of hours and therefore this operation should take place once and in advance to the interactive session. Data mining services based on supervised and unsupervised learning with processing on worker nodes and on the databases side will be elaborated. A typical run takes from several hours to several days. In an interactive work with datasets typical for LHC (Large Hydron Collider), the objective of training the neural network in a few minutes results in the distribution of the work across $o(10 - 100)$ Grid nodes.

2.4 Air Pollution and Weather Forecasting

This application includes sea wave modelling with the Coupled Ocean/Atmosphere Mesoscale Prediction System, data mining applied to downscaling weather forecasts, an implementation of air-pollution models [6]. The application is data- and compute-intensive and requires efficient management and transfer of gigabytes. A large community shares massive data sets and institutions often create local copies and replicas of the simulated and observed data sets to overcome long wide-area data transfer latencies. The data management should not only provide security services, but also contain tools able to determine where all the existing copies are located, to enable a user to decide whether to access existing copies or create new ones to meet the performance needs of their applications.

3 Application Specific Grid Services

Application specific grid services are new services which enable interactive execution of applications on the Grid and are designed for the CrossGrid applications; these are: the User Interaction Services and the Grid Visualization Kernel.

3.1 User Interaction Services

The User Interaction Services should enable control of data flow between various Grid components connected together for a given interactive application. It should also synchronize the simulation and visualization, and - according to the required type of simulation - this service should enable a plug-in mechanism for the required components. Software interfaces to the User Interaction Services are presented in Fig. 1. Together with the CrossGrid scheduler, this module will cooperate with underlying resource managers and information services: resource broker (both DataGrid [7] and Globus [12]), Condor-G [16] system for High Throughput Computing, Nimrod/G [5] tool for advanced parameter study, Grid monitoring service and Globus GIS/MDS. When planning the detailed solution we will profit with Cactus experience [1]. The User Interaction Services should make it possible to start complex interactive applications and to steer the simulations. During the simulation the user should be able to cancel a job after an inspection of its output. The interactive application should also be restarted immediately. The service should enable an advance reservation and the use of priorities.

3.2 Grid Visualization Kernel

The Grid Visualization Kernel (GVK) will interconnect distributed simulation with visualisation clients. It will allow interactive, near real-time visualization for Grid applications on arbitrary visualization devices.

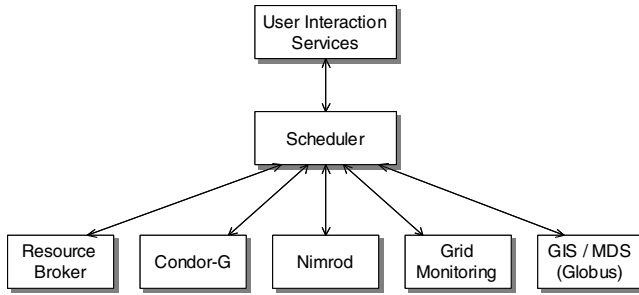


Fig. 1. User Interaction Services and other Grid components

4 New Generic Grid Services

To this category belong new Grid services with universal features, like resource management, roaming access, optimization of data access, and monitoring [6].

4.1 Scheduler

The CrossGrid Resource Management System will be based on self-adaptive scheduling agents. The scheduling agents will decide where, when, and how to run parallel jobs on Grid resources as specified by a set of policies, priorities, requirements and limits. The scheduler is responsible for deciding on the allocation of application tasks to computing resources in a way which guarantees that applications are conveniently, efficiently and effectively executed. This scheduler will be an extension of the DataGrid Workload Manager System. It will automate most of the activities related to resource discovery and system selection. The scheduler will manage two types of jobs: jobs consisting of single applications (sequential or parallel) and composite jobs consisting of collections of single applications (each of which can be either sequential or parallel) which exhibit some sort of inter-dependency. The applications will be described using a specific job description language.

4.2 Roaming Access

The Roaming Access that will allow the user to access his environment from remote computers consists of a Migrating Desktop, the application portals and the Roaming Access Server which manages user profiles, user authorization and authentication and job submission. These features are available through a user interface (see Fig. 2).

4.3 Optimization of Data Access

The Optimization of Data Access Service consists of a Component-Expert System, a Data-Access Estimator and a GridFTP Plugin. The Component-Expert

System is a heuristic system for selecting of components (data handlers) in a way which allows flexible and optimal data access. It optimizes the usage of the storage systems through proper selection of storage strategies due to different applications' requirements and existing storage capabilities. A human expert should prepare the decision rules. The Data Access Estimator estimates the bandwidth and the latency of data availability inside a Storage Element and this anticipation is used by the Replica Manager to take its decisions. The estimation of the data-arrival latency is different for secondary and tertiary storage systems and for databases. The GridFTP Plugin is designed to expand the capabilities of the GridFTP server according to requirements of the Component-Expert System. The interaction of the Optimisation of Data Access Service with other CrossGrid components is presented in Fig. 3.

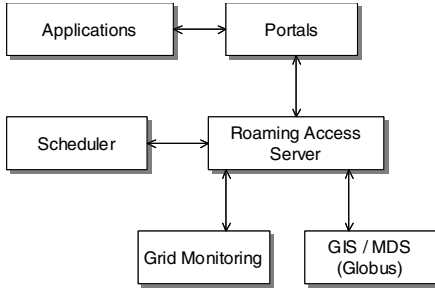


Fig. 2. Interaction of the Roaming Access Server with other Grid components

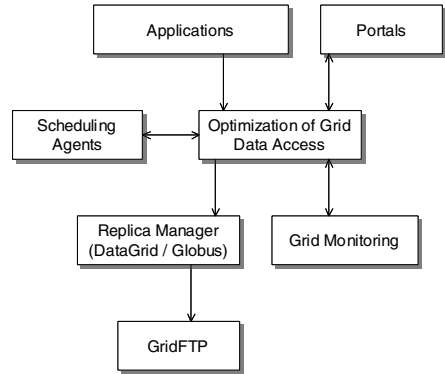


Fig. 3. Optimization of data access and other Grid components

4.4 Monitoring System

The CrossGrid Monitoring System works by delivering the low-level data intended for automatic extraction of high-level performance properties and on-line tool support, oriented towards application-bound activities, as well as data for network-oriented activities on the Grid. The Monitoring System consists of an OMIS-based application monitoring system OCM-G [2], additional services for non-invasive monitoring, Jiro-based services for Grid-infrastructure monitoring.

5 New Fabric Components

The new components are designed for optimization of local data access and for instrument management. The software which enables optimization of access to local data is called the Tape Resident Large Files middleware. This middleware is placed on top of an existing hierarchical storage system. It is intended to speed up read access to large files (more than 100 MB) stored on tapes, by means of reducing latency times. Within the framework of the CrossGrid Project we consider developing an application-oriented Resource Manager for instruments like radars which provide input for the flood crisis team support application.

6 Tools for Application Development

The CrossGrid application programming environment consists of:

- an MPI code debugging and verification tool to detect and report erroneous use of MPI communication functions,
- a Grid benchmark suite to gauge the performance of a given Grid configuration,
- a performance evaluation tool to support measurement and prediction of an application's performance on the Grid [3].

The possible interfaces between this tool environment and other Grid services are given in Fig. 4.

The portal will provide a unified and consistent window on the CrossGrid environment. The Grid tools will be integrated into the portal providing easier access for users of the Grid applications. The CrossGrid portals will securely authenticate and authorize users and it will allow them to view pertinent resource information obtained and stored in a remote database. The Migrating Desktop will offer a transparent user work environment, independent of the system version and hardware and enable users to access Grid resources and local resources from remote computers independently of their localization and terminal type.

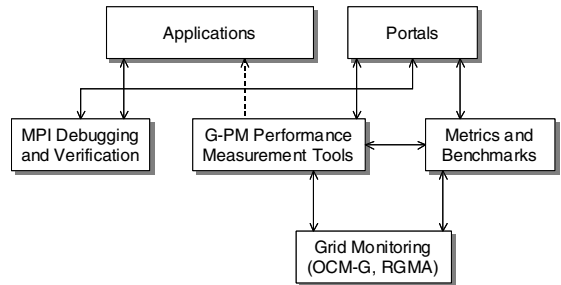


Fig. 4. The application programming environment and grid services

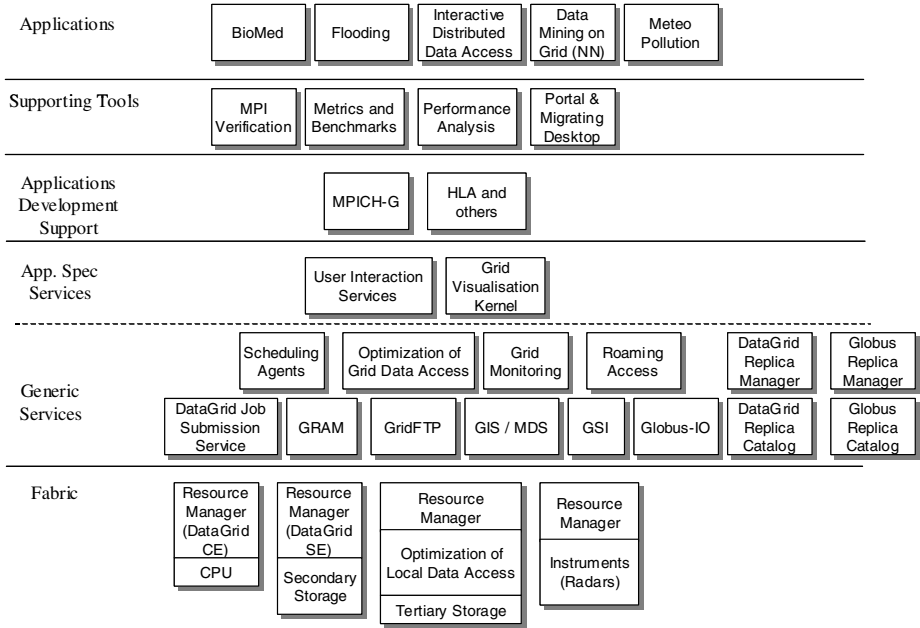


Fig. 5. General overview of the CrossGrid architecture

7 Architecture Overview

Fig. 5 presents the current version of the general overview of the CrossGrid architecture. The software components which will be developed within the framework of the Project and those from Globus [12] and DataGrid [7] are organized in layers similarly to the approach presented in [9]. We were also taking into account the architecture of the DataGrid [15], and recent results of the GLUE [13].

The User Interaction Services will cooperate with the HLA [14] module, the main tasks of which will be to maintain a preselected ordering of simulation events and to provide specific communication primitives. We are currently investigating two solutions: porting the HLA system to the Grid environment, and developing a Grid-oriented module which would encapsulate HLA functionality.

Each CrossGrid application may require specific configuration of the Grid services and tools. All applications of the CrossGrid require authentication and authorization as every action taken either by user (through the portal), or by any software component has to be secure. Therefore we will use Globus and DataGrid security systems as essential components of the CrossGrid architecture.

8 Concluding Remarks

This paper presents an overall view of the CrossGrid architecture, its components, their functionality and relations between them as well as with components

from Globus and DataGrid projects. The components of the CrossGrid software will be developed according to the iterative improvement approach consisting of fast prototyping, design, implementation, testing on testbeds, evaluation and further improvement. All software will be produced according to the evolutionary life-cycle model with well established phases of specification, development and validation. The key issue in this approach is collecting and improving the Software Requirements Specifications which are the starting point for the design of software architecture (on a high level), detailed design, implementation, and testing.

The most important open question is which technology should be used to efficiently implement the required functionality and how to organize interfaces between components. To solve this problem we must obtain more information and working experience with the recently proposed Open Grid Services Architecture [11, 10].

Acknowledgements

We are very grateful to M. Turała, M. Garbacz, P.M.A. Sloot, D. van Albada, L. Hluchy, W. Funika, R. Wismüller, J. Kitowski, and J. Marco for discussions and suggestions, and to P. Nowakowski for his comments.

References

- [1] Allen, G., Angulo, D., Foster, I., Lanfermann, G., Liu, C., Radke, T., Seidel, E., and Shalf, J.: The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *International Journal of High Performance Computing Applications* **15** (4) 345-358 (2001); <http://www.cactuscode.org> 18
- [2] Balis, B., Bubak, M., Funika, W., Szepieniec, T., and Wismüller, R.: An Infrastructure for Grid Application Monitoring. In *Proc. 9th EuroPVM/MPI Users' Group Meeting*, Linz, Austria, Sept. 2002, LNCS, Springer 2002 20
- [3] Bubak, M., Funika, W., and Wismüller, R.: The CrossGrid Performance Analysis Tool for Interactive Grid Applications. In *Proc. 9th EuroPVM/MPI Users' Group Meeting*, Linz, Austria, Sept. 2002, LNCS, Springer 2002 21
- [4] Bubak, M., Marco, J., Marten, H., Meyer, N., Noga, N., Sloot, P.M.A., and Turała, M.: GrossGrid - Development of Grid Environment for Interactive Presented at PIONIER 2002, Poznan, April 23-24, 2002, *Proceeding*, pp. 97-112, Poznan, 2002 16
- [5] Buyya, R., Abramson, D., and Giddy, J.: Nimrod-G Resource Broker for Service-Oriented Grid Computing. *IEEE Distributed Systems Online*, **2** (7) November 2001; <http://www.csse.monash.edu.au/~rajkumar/ecogrid/> 18
- [6] GrossGrid - Development of Grid Environment for Interactive Applications. <http://www.eu-crossgrid.org> 16, 17, 18, 19
- [7] DataGrid Project: <http://www.eu-datagrid.org> 17, 18, 22
- [8] Foster, I., Kesselman, C. (eds.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999 17

- [9] Foster, I., Kesselman, C., Tuecke, S. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications* **15** (3) 200-222 (2001); <http://www.globus.org/research/papers/anatomy.pdf> 22
- [10] Foster, I., Kesselman, C., Nick, J.M., and Tuecke, S.: The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration, January 2002, <http://www.globus.org> 23
- [11] Gannon, D., Bramley, R., Fox, G., Smallen, S., Rossi, A., Ananthakrishnan, R., Bertrand, F., Chiu, K., Farrellee, M., Govindaraju, M., Krishnan, S., Ramakrishnan, L., Simmhan, Y., Slominski, A., Ma, Y., Olariu, C., Rey-Cenvaz, N.: Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. Department of Computer Science, Indiana University. <http://www.extreme.indiana.edu/~gannon/> 23
- [12] Globus Project: <http://www.globus.org> 17, 18, 22
- [13] Grid Laboratory Uniform Environment: <http://www.hicb.org/glue/glue.htm> 22
- [14] High Level Architecture: <http://hla.dmsi.mil> 22
- [15] Jones, B. (ed.): DataGrid Architecture: <http://edms.cern.ch/document/333671> 22
- [16] Livny, M.: High Throughput Resource Management. In Foster, I., Kesselman, C. (eds.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 311-337, 1999 18

Application of Component-Expert Technology for Selection of Data-Handlers in CrossGrid

Lukasz Dutka¹ and Jacek Kitowski^{1,2}

¹ Institute of Computer Science, AGH
al. Mickiewicza 30, 30-059 Cracow, Poland

² ACC CYFRONET AGH
ul. Nawojki 11, 30-950 Krakow, Poland
{dutka,kito}@uci.agh.edu.pl

Abstract. In this paper the Component-Expert technology and its usage for data access problems in the CrossGrid environment are presented. The technology makes use of the component architecture supported by the expert system methodology in order to assist selection of the most appropriate component during the program operation. In the CrossGrid environment its purpose is to support local management of data, based on selection of components (data-handlers) for data storing and retrieving inside storage nodes.

1 Introduction

In the CrossGrid project [1] user applications are characterized by the interaction with a person in a processing loop. They require a response from the computer system to an action by the person in different time scales; from real through intermediate to long time, and they are simultaneously compute- as well as highly data-intensive. Examples of these applications are: interactive simulation and visualization for surgical procedures, flooding crisis team decision support systems, distributed data analysis in high-energy physics, air pollution combined with weather forecasting. To enable efficient development of this category of applications for the Grid environment efficient distributed data access is needed. The users should be able to run their applications on the Grid in an easy and transparent way, without detailed knowledge of the Grid structure and operation. Additionally, the way of data-access has to be as much efficient as possible but this way is tightly depended on the internal structure of a storage node or center, where required data are stored or retrieved. In many cases the nodes that keep and process data are different and often they are selected randomly by a job scheduler. Therefore, if an application lacks information on the internal structure of the storage node or data organization, then it cannot perform storing or retrieving data efficiently, due to data and storage node diversity. On the other hand, the implementation of data-serving solution for a storage node should be flexible and easy to tune.

In this paper the component-expert architecture is presented (shown roughly in [2]) with its implementation to grid enabled storage. Its flexibility resulting from automatic component selection according to their specialization is

discussed. Therefore, the approach is especially addressed for development of large-scale systems [3, 4, 5]. Although it originates from cluster computing it is worth to mention work in the area of component-based software engineering (see e.g. [6, 7]). Many models and component architectures have been developed, like COM/DCOM [8], CORBA [9, 10], JavaBeans [11, 12] and ActiveX [13]. The proposed architecture combines the component approach with the expert methodology in order to support the best component selection during the program operation. It extends in some sense the strategy of mass-storage management proposed in the framework of DataGrid Project [14].

2 Components for the Component-Expert Technology

In the reported approach a component consists logically of a header, a code, a stream of input parameters, a stream of output parameters and a data stream (Fig. 1). The header is composed of a component type and of an attributes list (Fig. 2).

The component-type describes the general purpose of the component. Examples of types for data-access problems are: **Read** for reading data components, **Write** for writing data components or **Estimate Latency** for components estimating latency of the data access.

The attributes list specifies a component specialization. The specialization of component represents the detailed purpose for which the component is best suited. The specialization is used by the rule-based expert system for deduction, i.e., for selection of the component from the component collection according to the context of component invocation (this context is also called 'call-environment' in the paper). Different types of the attribute values may be implemented, e.g., numbers, strings, enumerators, etc. Taking **Read** type component as a further example, there can be an universal component, which reads each kind of data, another which is specific for small files reading, and others – specialized for huge files, for remote reading using ftp or GridFTP protocols, for tape-resident data, etc.

The input parameters that are passed to the component are used by its code. The output parameters stream fetches low-volume results of the component

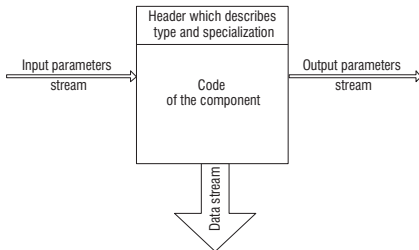


Fig. 1. Logical scheme of the component structure

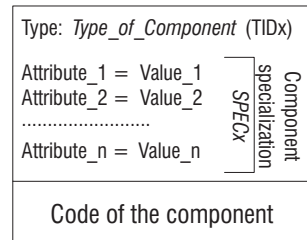


Fig. 2. Internal structure of the header

execution and can be used by other components or programs. The data stream carries high-volume data returned by the component code.

2.1 Implementation of Components of Component-Expert Technology for CrossGrid Environment

To make the discussion more specific, we present an application draft of the component-expert technology for data-access problem in the CrossGrid project. In the proposed solution each storage node operates its own set of components. The components in every set are categorized by types, which characterize the purpose of their usage, for example there can be components for data-transmission from storage elements, for estimation of some data access factors inside storage elements or for managing data. These kinds of usage directly imply a variety of components types (e.g., read, write, test throughput or predict latency of data availability, etc.). The complexity of the CrossGrid environment entails the need of diversification of components of each type. This diversification appears as a component specialization.

Several features are defined, which should be taken into account while developing the data-access system for grid environment:

- heterogeneous storage devices and environments, distributed over the grid (e.g., on-line, off-line, near-line, backup or hierarchical),
- different requirements concerning availability of different data types in different user localizations (e.g., quality-of-service or best-effort, replication strategies),
- heterogeneous storing strategy (e.g., raw or hierarchical files, relational or object databases),
- specific requirements for data storage (e.g., direct or sequential access).

Such aspects influence the set of attributes, which is used to describe the specialization of components. A sample set of attributes is presented below.

1. **User ID** – represents a unique identifier of the user. It allows one to build components specialized for a dedicated user.
2. **Data ID** – a unique data identifier for coupling component specialization with some kind of data.
3. **Required throughput** – represents a value of the required throughput. The requirement of the throughput often entails the need for special algorithms and/or procedures (e.g., flow control, monitoring throughput, etc.). Thus additional components are useful.
4. **Data type** – shows the type of data. For example the system can store or retrieve data for *Water surface models*, *LHC experiments* or *Indexed mpegs*. Such data types might require specialized components.
5. **Application purpose** – represents a purpose of the running application (e.g., *Medicine simulation*, *High energy physics* or *Flood crisis team support*).

6. **Storage type** – means a type of the storage system. There are many types of storage systems, e.g., disk drives, HSM storage systems, tapes, RAID systems, etc.
7. **Critical level** – represents the critical level of the running application. Sometimes the critical level implies the need for usage of special algorithms which could be represented by another group of components.

The attributes have been selected taking into account the nature of the data-access problem in the grid environment, which is characterized by distribution and diversity of equipment.

In the example of components specialization (see Fig. 3) the simplest **Read** type was chosen.

| <table><tr><th>Component: 1</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>Data type = Indexed mpeg</td></tr><tr><td>Application purpose = Medicine simulation</td></tr><tr><td></td></tr></table> | Component: 1 | Type = Read | Specialization | Data type = Indexed mpeg | Application purpose = Medicine simulation | | <table><tr><th>Component: 2</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>User ID = Tester</td></tr><tr><td>Data type = Indexed mpeg</td></tr><tr><td>Application purpose = Medicine simulation</td></tr><tr><td></td></tr></table> | Component: 2 | Type = Read | Specialization | User ID = Tester | Data type = Indexed mpeg | Application purpose = Medicine simulation | | <table><tr><th>Component: 3</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>User ID = Tester</td></tr><tr><td>Data type = LHC experiment</td></tr><tr><td>Application purpose = High energy physics</td></tr><tr><td></td></tr></table> | Component: 3 | Type = Read | Specialization | User ID = Tester | Data type = LHC experiment | Application purpose = High energy physics | |
|---|--------------|-------------|----------------|--------------------------|---|----------------------------|---|---|--------------|----------------|------------------|----------------------------|---|----------------------------|---|---|--------------|----------------|------------------|---------------------------------|---|--|
| Component: 1 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| Data type = Indexed mpeg | | | | | | | | | | | | | | | | | | | | | | |
| Application purpose = Medicine simulation | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| Component: 2 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| User ID = Tester | | | | | | | | | | | | | | | | | | | | | | |
| Data type = Indexed mpeg | | | | | | | | | | | | | | | | | | | | | | |
| Application purpose = Medicine simulation | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| Component: 3 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| User ID = Tester | | | | | | | | | | | | | | | | | | | | | | |
| Data type = LHC experiment | | | | | | | | | | | | | | | | | | | | | | |
| Application purpose = High energy physics | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| <table><tr><th>Component: 4</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>Storage type = RAID</td></tr><tr><td>Required throughput = True</td></tr><tr><td>Data type = LHC experiment</td></tr><tr><td></td></tr></table> | Component: 4 | Type = Read | Specialization | Storage type = RAID | Required throughput = True | Data type = LHC experiment | | <table><tr><th>Component: 5</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>Required throughput = True</td></tr><tr><td>Data type = Water surface model</td></tr><tr><td>Critical Level = Very high</td></tr><tr><td></td></tr></table> | Component: 5 | Type = Read | Specialization | Required throughput = True | Data type = Water surface model | Critical Level = Very high | | <table><tr><th>Component: 6</th></tr><tr><td>Type = Read</td></tr><tr><td>Specialization</td></tr><tr><td>Data type = Water surface model</td></tr><tr><td>Storage type = HSM</td></tr><tr><td></td></tr></table> | Component: 6 | Type = Read | Specialization | Data type = Water surface model | Storage type = HSM | |
| Component: 4 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| Storage type = RAID | | | | | | | | | | | | | | | | | | | | | | |
| Required throughput = True | | | | | | | | | | | | | | | | | | | | | | |
| Data type = LHC experiment | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| Component: 5 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| Required throughput = True | | | | | | | | | | | | | | | | | | | | | | |
| Data type = Water surface model | | | | | | | | | | | | | | | | | | | | | | |
| Critical Level = Very high | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |
| Component: 6 | | | | | | | | | | | | | | | | | | | | | | |
| Type = Read | | | | | | | | | | | | | | | | | | | | | | |
| Specialization | | | | | | | | | | | | | | | | | | | | | | |
| Data type = Water surface model | | | | | | | | | | | | | | | | | | | | | | |
| Storage type = HSM | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | |

Fig. 3. Sample subset of components type **Read** with their specializations

3 Component-Expert Architecture

In the Component-Expert architecture the most important modification in comparison with the classical one is transfer of responsibility for proper components selection from the programmer to the system, which has got some kind of 'intelligence'. For this reason elements of the expert systems [15, 16, 17] are introduced.

To realize this strategy the system consists of the following elements (see Fig. 4): Components, a Components Container, an Expert Component Management Subsystem, an Expert System and an optional Knowledge Base. The Components Container keeps in groups components of the same type – **TID**, but with different specializations – **SPEC**. Flexibility in component invocation enables different combinations of the component type – **TID**, and the call-environment

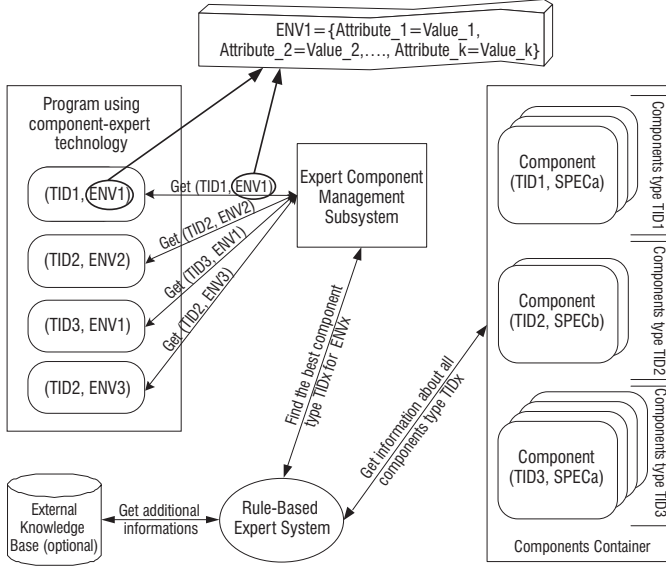


Fig. 4. Connection Diagram of Component-Expert Application

– ENV. The call-environment, that represents context of invocation, facilitates proper selection of components (cf. Figs. 3 and 4).

The operation of the Component-Expert system is summarized as follows:

1. The program requires service for the $TIDx$ component type for the $ENVy$ call-environment (see section 3.1). It requests from the Expert Component Management Subsystem a handle to the particular component, most suitable for $ENVy$.
2. The Expert Component Management Subsystem parses obtained information and using the Expert System it searches the best component for $ENVy$.
3. The Expert System gets information about all $TIDx$ type components from the Components Container.
4. Based on the rule-based knowledge of the expert system and on the External Knowledge Base, as well as on the specialization of components of the required type, the Expert System takes a decision, which component is most suitable for $ENVy$.
5. If the Expert System finds an appropriate component then the Expert Component Management Subsystem returns a handle to the component, otherwise an error message is returned.
6. The program exploits the component referencing the handle.

3.1 Call-Environment

An important element of the whole architecture is the Call-Environment, which defines requirements and expectations that should be fulfilled by the most suit-

able component for the invocation context. The call-environment consists of the attributes list with their values (Fig. 4). It is connected with the required component type and it appears at the program side only. The pair component-type and call-environment is passed to the Expert Component Management Subsystem, which parses and processes it. Every call of the component in the whole application may have different call-environments. At the component invocation the most suitable component of a given type is to be selected from all components accumulated in the system. The precision of the component fitting depends on precision of the call-environment description, specialization of the components and knowledge bases.

For the data-access problem, examples of the call-environment are the following:

```
ENV1={User ID = Tim; Data ID = file1.dat; Data Type = Indexed Mpeg;
Application Purpose = Medicine Simulation}
ENV2={User ID = Tester; Data ID = file1.dat; Application Purpose =
Medicine Simulation} ENV3={User ID = Tester; Data ID = file2.dat}
ENV4={User ID = Tester; Data ID = file3.dat; Required throughput=768 kb;
Critical level = Very high}
ENV5={User ID = Jim; Data ID = file3.dat; Critical level = Very high}
ENV6={User ID = Jim; Data ID = file4.dat}
```

The internal structure of the call-environment and the specialization of components are similar, although their meanings are different. The call-environment is a kind of list, which is used for description of requirements. The specialization of components is another list to describe the detailed purpose of the component. A hyphen between them is the Expert System (see section 3.2).

3.2 Expert System

The rule-based Expert System [15, 16] holds knowledge about the whole system, introduced by a human expert, whose duty is to set a hierarchy of importance of components.

The Expert System selects a component of the defined type according to the specified call-environment. The most important factors taking into account during the selection process are knowledge, which is represented by means of rules and actual specializations of components that are kept in the Components Container. Some kinds of problems could need additional deduction, therefore implementation of external (optional) knowledge bases would extend the information given by the call-environment and improve the component matching process.

In the case of the CrossGrid project the set of the rules will be changed rarely and by a human-expert only. Therefore, no possibility of their automatic modification is proposed due to difficulty in definition of metrics for selection quality. However, the knowledge gathered in external knowledge bases will be upgraded continuously, thus the whole solution will not be static.

The Expert System should store some logs, which may be useful for a human expert in the system tuning.

4 Example of Session in CrossGrid Environment

An example of the data-access session is presented below. The description of the attributes is stated in section 2.1. In real implementations they become more complicated.

1. An application requests data from some data-object. This implies that it requires the usage of the best component type **Read** for storage element that keeps the required data. Since the call-environment is needed, the attributes **Data ID** and **User ID** are defined and additionally attributes **Required throughput**, **Application purpose** or **Critical level** are set.
2. The pair consisting of the component type and the call-environment is sent to the storage element, which passes it to the Expert Component Management Subsystem.
3. The current call-environment is extended using External Knowledge Base, thus values of **Data type** and **Storage type** attributes are found.
4. The Expert System finds the best component for the given call-environment to be used by the application.

5 Implementation

The presented component-expert approach has already been successfully implemented in the domain of web-based, large-scale information systems in Windows NT environment [2] using Microsoft ASP technology [18]. In the case of the CrossGrid project, component expert-architecture is being realized by the dedicated module – the Component-Expert System – written from the scratch using the C++ language and optimized for the UNIX environment.

The component structure presented in Figs. ?? and 2 represents its logical scheme only. In practical realization it is being implemented to avoid any parser usage. Exchange of rules of the Expert System on-the-fly during the run-time is planned, while another set is required.

6 Conclusions

The application of the component-expert architecture, especially for the CrossGrid environment results in some advantages, like minimizing programmer responsibility for component choice, ease of programming in heterogeneous environment, internal simplicity of components code, increase efficiency of the programming process and ease of system extension.

The storage nodes using the component-expert technology will be flexible and easy to manage. Programmers will obtain the powerful data-access environment, easy to building efficient applications.

Acknowledgement

The work described in this paper was supported in part by the European Union through the IST-2001-32243 project "CrossGrid". AGH Grant is also acknowledged.

References

- [1] CrossGrid – Development of Grid Environment for interactive Applications, EU Project, IST-2001-32243. 25
- [2] Ł. Dutka and J. Kitowski, Expert technology in information systems development using component methodology, in: Proc. of Methods and Computer Systems in Science and Engng. Conf. Cracow, Nov.19-21, 2001, pp. 199-204, ONT, Cracow, 2001 (in Polish). 25, 31
- [3] P. Allen and S. Frost, Component-Based Development for Enterprise Systems: Applying the Select Perspective, Cambridge Univ. Press, 1998. 26
- [4] P. Herzum and O. Sims, Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise, John Wiley & Sons, 1999. 26
- [5] A. W. Brown, Large-Scale, Component Based Development, Prentice Hall, 2000. 26
- [6] A. M. Zaremski and J. M. Wing, Specification matching of software components, ACM Trans. Software Engng. and Methodology, 6 (4) (1997) 333. 26
- [7] L. Iribarne, J. M. Troya and A. Vallecillo, Selecting Software Components with Multiple Interfaces, Proc. of the 28th Euromicro Conf., Dortmund, Sept. 2002, IEEE Computer Society Press (to be published). 26
- [8] R. Rock-Evans, DCOM Explained, Digital Press, 1998. 26
- [9] D. Slama, J. Garbis and P. Russell, Enterprise Corba, Prentice Hall, 1999. 26
- [10] Z. Tari and O. Bukhres, Fundamentals of Distributed Object Systems: The Corba Perspective, John Wiley & Sons, 2001. 26
- [11] R. Monson-Haefel, Enterprise JavaBeans, O'Reilly & Associates, 2001. 26
- [12] P. G. Sarang, et al, Professional EJB, Wrox Press, Inc., 2001. 26
- [13] A. T. Roff, ADO: ActiveX Data Objects, O'Reilly & Associates, 2001. 26
- [14] J. C. Gordon, O. Synge, J. Jensen, T. Eves, G. Zaquine, Mass Storage Management, Cern Internal Report, DataGrid-05-D5.2-0141-3-4, 2002. 26
- [15] D. T. Pharm, (ed.), Expert System in Engineering, Springer-Verlag, 1988. 28, 30
- [16] C. T. Leondes, (ed.), Fuzzy Logic and Expert Systems Applications, Academic Press, 1998. 28, 30
- [17] I. Vlahavas and N. Bassiliades, Parallel, Object-Oriented and Active Knowledge Base Systems, Kluwer Academic Publishers, 1998. 28
- [18] <http://www.aspin.com/>. 31

Training of Neural Networks: Interactive Possibilities in a Distributed Framework

O. Ponce¹, J. Cuevas², A. Fuentes³, J. Marco¹, R. Marco¹,
C. Martínez-Rivero¹, R. Menéndez⁴, and D. Rodríguez¹

¹ Instituto de Física de Cantabria (CSIC-UC)

Avda. Los Castros s/n, 39005 Santander, Spain

² Facultad de Ciencias, Avda. Calvo Sotelo s/n, 33007 Oviedo, Spain

³ RedIris CSIC, C/ Serrano 142, Madrid, Spain

⁴ Dpto. Electrónica y Computadores, E.T.S.I.I.T.

Avda. Los Castros s/n, 39005 Santander, Spain

Abstract. Training of Artificial Neural Networks in a Distributed Environment is considered and applied to a typical example in High Energy Physics interactive analysis. Promising results showing a reduction of the wait time from 5 hours to 5 minutes obtained in a local cluster with 64 nodes are described. Preliminary tests in a wide area network studying the impact of latency time are described; and the future work for integration in a GRID framework, that will be carried in the CrossGrid European Project, is outlined.

1 Introduction

The GRID [1] framework provides access to large shared computing resources distributed across many local facilities.

High-throughput computing suits this approach: as an example, in the High Energy Physics (HEP) field thousands of independent simulation batch jobs, corresponding to a physics channel, are processed distributed across computing sites in France, UK, Italy, Spain, etc., and the resulting datasets integrated in a centralized database at CERN. In contrast, we consider here another complex computing problem in HEP that does not allow independent data processing and typically seems better suited for a high-performance environment: interactive HEP data analysis using Artificial Neural Networks (ANN).

ANN have become the “de facto” standard in HEP to address complex multidimensional analysis problems. As a relevant example, the recent search for the Higgs Boson at the Large Electron Positron Collider (LEP) at CERN used this technique in most of the channels, obtaining a significant performance improvement compared to traditional “sequential cuts” or “likelihood-based” analysis. Typically, these networks are trained on large samples of simulated events, but the network structure itself is moderately complex: in the DELPHI collaboration the Higgs search [2] in the hadronic channel used an ANN with 16 inputs, two intermediate layers of 10 hidden nodes each, and one output node (noted 16-10-10-1 architecture). Training typically requires 1000 epochs, running on samples

of about 500000 events pre-selected from total simulated samples of around 5 Million events. Independent samples of the same size are used to test the ANN. As no specific rules exist to build an optimal ANN, different architectures and input variables are used in the process to get the optimal final function. This demands a large computing power, because each training process requires from hours to days of a current PIII 1GHz CPU. However, physicist would like to use them in an interactive way, without having to wait so many time to try out different possibilities for the analysis. A much faster training of an ANN can be achieved either by using a much more powerful computer, or by running in a distributed way, either in a parallel machine, in a local cluster, or, to get more resources at the same time, at GRID scale.

The possibility of parallelizing ANN (see [3] for a good survey), with several approaches, has been studied for a long time. We follow an approach based on data partition, and have implemented the program in commodity hardware. A similar approach in a Beowulf cluster can be found in [4]. This article presents, in the first place, the work done to implement a distributed technique to train ANN in a local cluster. The plans and first steps taken to move it into the GRID framework are then outlined. This final part of the work will take place inside the CrossGrid European Project [5], and will be used for HEP interactive applications and with a similar approach in the meteorology field using Self Organizing Maps [6].

2 Distributed Training of a Neural Network Using MPI and Load Balancing through Data Partition

We will address the following well defined problem from the past Higgs Boson search at LEP:

Given a sample of 644577 simulated events with 16 relevant variables per event, 20000 of them corresponding to signal events, train a neural network with a 16-10-10-1 architecture with an output node returning 1 for the Higgs signal and 0 for the background, using 1000 epochs typically, and with a test sample of similar size.

The ANN is described by its architecture, that also fixes the number of involved weights, or parameters, that have to be optimized. A weight is needed to connect each pair of nodes (see figure 1), so the total number of weights for our example is 270. The objective of the training is the minimization of the total error, given by the sum of errors for all events, each one defined as the quadratic difference between the ANN output computed corresponding to each event and the 1 or 0 value corresponding to a signal or background event. The minimization procedure is iterative, each iteration being called an epoch. There are several methods, but only some of them can be adapted for distributed computing. We will use here the BFGS method [7], a gradient descent where errors and gradients at each step are additive, so it allows their calculus at different computing nodes, and their addition by a master node to prepare the next iteration step. Iteration is repeated until the minimization error does not

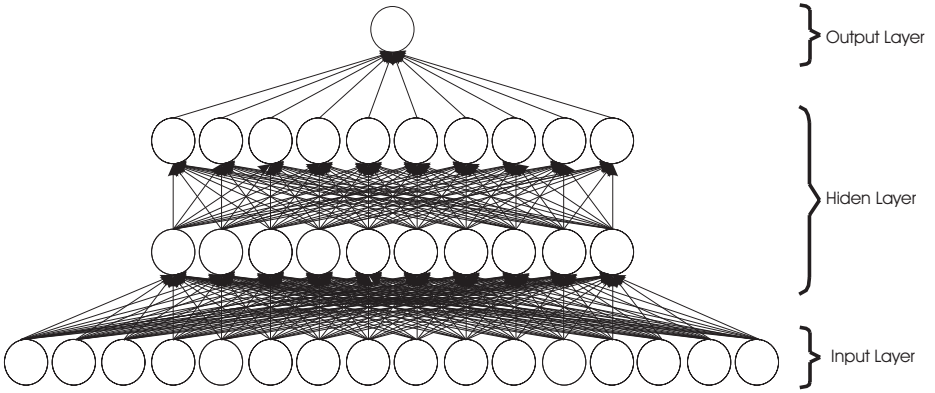


Fig. 1. 16-10-10-1 ANN

vary significantly. As no safe rule is known to avoid local minima, the procedure should be repeated with different values for the initial weights, and convergence checked in the training and test sample to the same error rate. Overtraining is not possible as the number of weights involved is significantly smaller than the number of events used for the training, and also would not affect the final physics results that are based only on the test sample.

The procedure to run in distributed mode this training process presumes a master node and several (N) slave nodes:

- i) the master node starts the work reading the input parameters for the ANN architecture, and setting the initial weights to random values
- ii) the training data is split into several datasets of similar size (taking into account the computing power of each node, to assure load balancing by data partitioning) and distributed to each slave node.
- iii) at each step, the master sends the weight values to the slaves, which compute the error and the gradient on the partial dataset and return them to the master; as both are additive, total errors are calculated by the master that prepares the new weights along the corresponding direction in the multidimensional space and updates the weights. The error computed in a separated test dataset is used to monitor the convergence.
- iv) the previous step (an epoch) is repeated until a convergence criteria is satisfied or a predetermined number of epochs is reached. The master returns as output the ANN weights, to prepare the corresponding multidimensional function.
- v) the whole training can be repeated with different initial random weights. The final error can be used to select the best final ANN function if different minima are found.

The ANN package that has been adapted is the very same used in the Higgs Boson search, namely the MLPfit package [8]. The program was used previously

in the framework of the PAW [9] interactive program for HEP. Typical training using this optimized program takes 22000 seconds in a PIII at 1 GHz.

The program, written in C language, has been adapted to distributed calculus using MPI [10]. In the current configuration, the master sends collective calls to the slaves to distribute the weights and return the errors.

3 Results in a Local Cluster

The program was initially tested with a single computing node acting as master and slave: an Intel IA32 box running linux kernel 2.4.5 with the gcc 2.7 compiler¹. The reference sample had 644577 events, with 16 variables each, generated by the Pythia generator [11], corresponding to Standard Model events in e+e- collisions at a center-of-mass energy of 205 GeV for the background, and signal events corresponding to a Standard Model Higgs Boson with mass 115 GeV/c^2 .

A 1000 epochs training of the ANN with 16-10-10-1 architecture takes approximately 5 hours in the reference computing node. The final error and weights corresponding to the given random set of initial weights were stored for later checks.

Next step was the distributed implementation, using MPI clients on machines different to the master node. Communication from Master to slaves was done through a “broadcast” operation sending 2.11 KB, corresponding to 270 weights; and the answer with a “reduce” operation, with similar message size, corresponding to the computed error and gradients.

Several factors were expected to have an impact on the total computing time: the need of a good load balancing and the time to transfer the weights from the master to the slaves, and return back the error. In particular, latency and bandwidth appear as critical depending on the computing time at each node and the size of the weights set. In this first example, the size was small (2.11 Kbytes) and the approximately 20 seconds time per epoch was expected to be partitioned between 2-64 machines, so at least a minimum of 300 ms of computing time was expected per slave machine for each epoch.

The first distributed tests were run in a local cluster with 80 nodes equal to the reference one. The “Santander GRID Wall”, SGW [12] is a local cluster of IA32 nodes as described previously, and interconnected via fast ethernet using three fast ethernet switches stacked with 2 GB/s. Several configurations were tested with 8, 16, 32 and 64 nodes.

The results of these tests were encouraging: thanks to the low latency and the reduced message size, the scaling with the number of nodes followed approximately an $1/N$ law. Running with 64 nodes can roughly reduce the wait time for a physicist doing interactive analysis from 5 hours to 5 minutes. Figure 2 shows the speedup function for these tests.

More complex strategies are needed to run in a more realistic cluster configuration, where the nodes are not reserved exclusively for this task and one user;

¹ An IBM x220 server, Pentium III at 1.26 GHz, 512 Kb cache with 36 GB SCSI disk and 640 MB RAM running RedHat Linux 7.2

Table 1. Time vs. number of slaves in local cluster (SGW) for 16-10-10-1 and 16-50-50-1 architectures

| Num. slaves | Total Time (s) | |
|-------------|--------------------------|-------------------------|
| | 16-10-10-1 (1000 epochs) | 16-50-50-1 (100 epochs) |
| 1 | 17622 | 16150 |
| 8 | 2240 | 2129 |
| 16 | 1123 | 1137 |
| 32 | 578 | 648 |
| 64 | 320 | 416 |

for example, when part of the nodes are running a long Monte Carlo simulation batch. This question will be addressed in some future work.

Another important point is the scaling with the neural network size. This is known to increase with the total number of weights. For example, using 644577 events to train an ANN with a 16-10-10-1 architecture (270 weights) in a cluster with a master and a slave needs 21 seconds per epoch. The same with a 16-50-50-1 architecture (3350 weights) takes 144 seconds per epoch.

There are two different problems to consider: first, as the size of the NN grows, the slaves need to perform more operations to get the gradient and the error for its set of events; second, the master has to work with a bigger matrix to decide the new weights in the next epoch. This is a double precision numbers square matrix of dimension equal to the number of weights: in the first case the matrix size in memory is 569.5 Kbytes, while in the second case it is 85.6 Mbytes.

The first problem can be solved adding more slaves, so the load is reduced for each single slave, or using faster slaves; but the second problem is far more complex because while the master is calculating the new weights for the next epoch, the slaves are idle. So, in this case the scaling of the execution time is not $1/N$. In fact, the master's computing power is now critical, not only the speed and architecture of the processor, but also the amount of memory of the machine, because swapping to disk increases largely the time employed in the matrix computations (up to a factor ten). This can be solved with the parallelization of the matrix computations, according to the number of weights.

4 From Local to Wide Distributed Computing

The use of nodes in a computing GRID is a very interesting possibility, but also poses many challenges. Most of them will be addressed using the existing GRID middleware like the MPI version MPICH-G2 in the Globus suite. Others are being studied in different projects, and in particular in the CrossGrid project. We will describe here only the first ideas about one of the main open questions: the communication latency issue. We expect latency times to vary from tenths to hundredths of milliseconds between two nodes well interconnected in a national

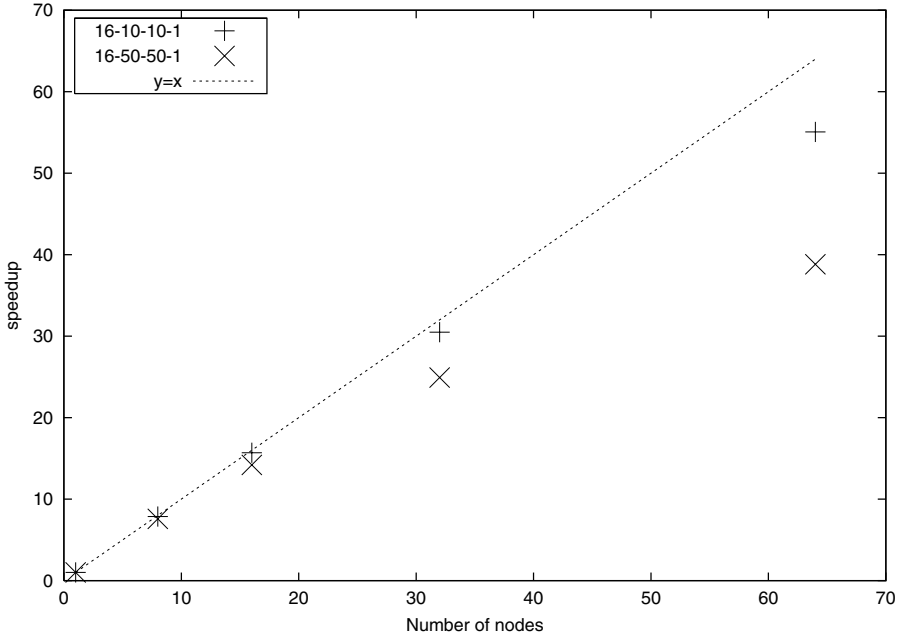


Fig. 2. ANN speedup in local cluster for 16-10-10-1 and 16-50-50-1 architectures

academic network, and similar ones for two nodes joined through the Géant [13] European network. To estimate the effect of the communication latency in the execution time of one simulation of the neural network program we have done two different kinds of tests. Both of them included two national nodes, the reference one already described in Santander, and another one in RedIris at Madrid.

In our first test we confirmed the performance difference between the computational nodes that we have used, a PIII 666 MHz at Madrid, and the PIII 1.26 GHz at Santander. As expected the performance of the PIII 1.26 GHz doubles the one of the PIII 666 MHz one.

Our second test was directed to evaluate the impact of the communication latency. For this test we used two different configurations to train the same simulation set (10072 events with 16 input variables, corresponding roughly to the 1/64th of the 644577 we have before). The first one was in a cluster with one master and two slaves, which were PIII 1.26 GHz, while the second configuration had two machines, a master and a slave, both of them PIII 1.26 GHz, in the same location (Santander, in a cluster with low communication latency between these nodes) and a PIII 666 MHz as slave in an external cluster at Madrid.

Tables 2 and 3 show the results obtained with the two network architectures and a different number of epochs (1000 or 100) for different network architectures showing the communication latency effect.

Table 2. Cluster configuration: three PIII 1.26 GHz nodes (a master and two slaves) in a local cluster. Total latency is defined as $\langle latency * MPIcalls * epochs \rangle$. Latency is obtained with the ping command

| ANN | Epochs | Total time (s) | Total latency (s) |
|------------|--------|----------------|-------------------|
| 16-10-10-1 | 1000 | 282 | 0.5 |
| 16-50-50-1 | 100 | 386 | 0.05 |

Table 3. Two PIII 1.26 GHz nodes: a master and a slave in a cluster, and an external PIII 666 MHz slave. Modelled time includes the different processing power factor for the slave plus total latency, and the time taken by the master node for matrix computation. The first three results correspond to three different network conditions

| ANN | Epochs | Modelled Time (s) | Total Time (s) | Total latency (s) |
|------------|--------|-------------------|----------------|-------------------|
| 16-10-10-1 | 1000 | 640 | 664 | 70 |
| 16-10-10-1 | 1000 | 724 | 721 | 140 |
| 16-10-10-1 | 1000 | 1240 | 1270 | 570 |
| 16-50-50-1 | 100 | 694 | 702 | 7 |

The main conclusion is that we need a load balancing mechanism when mixing nodes with different performance and latency time, because this is directly reflected on the total time. This can be achieved using different training sample sizes at each node. To decide the different sample sizes we will use in each node, we have used the following modelling: in our two slaves case we, as first approach, assigned a load proportional to the CPU speed. A better estimation could be done considering the latency effect also.

Load balancing avoiding dead times at nodes local to the master is possible. As an example, Table 4 shows how an increase in global performance of about 50% can be reached balancing the training sample.

5 Conclusions and Future Work

The results shown in this paper proof the feasibility of running a distributed neural network in a local cluster reducing the wait time for a physicist using this tool from hours to minutes. They also indicate that this approach could be extended to a GRID environment, with nodes distributed across Internet, if the latency is low enough.

Several points need further work:

- i) the parallelization of the matrix computations to obtain the new weights for large neural networks.
- ii) tests with more distributed nodes across a larger testbed and check of basic practical issues like how to address a dead node in the collective MPI call

Table 4. Test of load balancing with an external node

| ANN | Epochs | Number of Events | | Total time (s) | Total latency (s) |
|------------|--------|------------------|--------------|----------------|-------------------|
| | | PIII 1.26 GHz | PIII 666 MHz | | |
| 16-10-10-1 | 1000 | 10072 | 10072 | 664 | 70 |
| 16-10-10-1 | 1000 | 20113 | 10072 | 695 | 80 |
| 16-50-50-1 | 100 | 10072 | 10072 | 702 | 7 |
| 16-50-50-1 | 100 | 20113 | 10072 | 701 | 8 |

iii) integration with the GRID framework: the use of MPICH-G2 has been tried in the local cluster, and will be extended to the GRID testbed.

All these topics will be addressed inside the CrossGrid project in the next months. Also different neural network algorithms, like Self Organizing Maps (SOM) implying the use of larger datasets, and including direct access to distributed databases, will be studied.

Acknowledgements

This work has been mainly supported by the European project CrossGrid (IST-2001-32243), and the Spanish MCyT project FPA2000-3261. Special thanks are given to CSIC for funding the Santander GRID Wall facility, and to the DELPHI Collaboration where part of the authors implemented the ANN for the Higgs Boson search.

References

- [1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999 [33](#)
- [2] DELPHI Collaboration. *Search for the standard model Higgs boson at LEP in the year 2000*. Phys. Lett. B 499:23-37, 2001 [hep-ex/0102036] [33](#)
- [3] Manavendra Misra. *Parallel Environments for Implementing Neural Networks*. Neural Computing Survey, vol. 1., 48-60, 1997 [34](#)
- [4] D. Aberdeen, J. Baxter, R. Edwards. *98c/MFLOP Ultra-Large Neural Network Training on a PIII Cluster*. Proceedings of Supercomputing 2000, November 2000 [34](#)
- [5] CrossGrid European Project (IST-2001-32243). <http://www.eu-crossgrid.org> [34](#)
- [6] Kohonen, T. *Self-Organizing Maps*. Springer, Berlin, Heidelberg, 1995 [34](#)
- [7] Broyden, Fletcher, Goldfarb, Shanno (BFGS) method. For example in *Practical Methods of Optimization* R.Fletcher. Wiley (1987) [34](#)
- [8] MLPFIT: a tool for designing and using Multi-Layer Perceptrons. <http://schwind.home.cern.ch/schwind/MLPfit.html> [35](#)
- [9] Physics Analysis Workstation. <http://paw.web.cern.ch/paw/> [36](#)
- [10] MPICH. <http://www-unix.mcs.anl.gov/mpi/mpich> [36](#)
- [11] T. Sjöstrand, *Comp. Phys. Comm.* 39 (1986) 347. Version 6.125 was used [36](#)
- [12] Santander GRID Wall. <http://grid.ifca.unican.es/sgw> [36](#)
- [13] Géant. <http://www.dante.net/geant/> [38](#)

An Infrastructure for Grid Application Monitoring^{*}

Bartosz Baliś¹, Marian Bubak^{1,2}, Włodzimierz Funika¹,
Tomasz Szepieniec¹, and Roland Wismüller^{3,4}

¹ Institute of Computer Science, AGH
al. Mickiewicza 30, 30-059 Kraków, Poland
{balis,bubak,funika}@uci.agh.edu.pl

² Academic Computer Centre – CYFRONET
Nawojki 11, 30-950 Kraków, Poland
szepieniec@icsr.agh.edu.pl

phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54

³ LRR-TUM – Technische Universität München
D-80290 München, Germany
wismuell@in.tum.de

phone: (+49 89) 289-28243

⁴ Institute for Software Sciences, University of Vienna
A-1090, Wien, Austria

Abstract. In this paper, we present a concept of the OCM-G — a distributed monitoring system for obtaining information and manipulating distributed applications running on the Grid. The purpose of this system is to provide a basis for building tools supporting parallel application development. We present Grid-specific requirements, propose a suitable architecture for the monitoring system, and describe the integration of the OCM-G with Grid services. OMIS is applied as an universal interface between the OCM-G and tools.

Keywords: Grid, monitoring, services, performance analysis, tools

1 Introduction

With the recent emergence of the Grid [8, 9, 10], application and tool-environment developers are faced with more complicated problems than in e.g. cluster computing. Grid resources are unprecedented both in amount and geographical distribution. The current solutions, mostly designed for clusters or supercomputers, cannot meet the scalability and efficiency requirements needed on the Grid. Tools for Grid environments are still at an early stage of development, however a foundation for a Grid tool environment has already been laid with the definition of a Grid Monitoring Architecture (GMA) [18]. This specification defines the monitoring architecture on the Grid in terms of the producer-consumer model.

^{*} This work was partly funded by the European Commission, project IST-2001-32243, CrossGrid [6].

A comparison of grid-oriented monitoring tools is given in [1]. The Globus Heartbeat Monitor [14] supports monitoring the state of processes. It reports failures of processes and provides notification of process status exception events. Network Weather Service [21] periodically monitors performance of networks and computational resources and forecasts this performance. The Information Power Grid project is developing its own monitoring infrastructure [20] which consists of sensors, actuators, and event services; XML is used to represent monitoring services. NetLogger [13] is designed for analysis of communication inside applications. GRM and PROVE [16] may be used for performance monitoring of applications running in distributed heterogeneous systems. Autopilot/Virtue [19, 17] is a performance measurement and resource control system. The DataGrid project [7] is developing monitoring systems oriented towards applications which operate with large data files; recently they proposed Relational Grid Monitoring Architecture which will be further developed within the Global Grid Forum.

The subject of this paper is an application monitoring system for the Grid called OCM-G. The OCM-G is meant to enable investigation and manipulation of parallel distributed applications running on the Grid, and provide a basis for building tools supporting parallel application development on the Grid.

In [4] and next, in [5], we considered alternatives for an architecture of the Grid application monitoring environment. This paper presents more detailed and mature concepts for our application monitoring system which is being developed in the framework of the CrossGrid project [6], where it will supply data to the programming tool environment [3]. This environment is being developed on the basis of our previous research [2] and on results of the APART project [11].

2 Monitoring of Grid Applications

Our approach differs from other approaches to Grid monitoring in several aspects. First, it is *application oriented* – it provides information about *running* applications. It works *on-line* – the information is not normally stored anywhere for later use but immediately delivered to its consumers (tools) and processed. The information is collected by means of *run-time instrumentation* which can be activated or deactivated on demand. Thus, data to be gathered can be defined at run-time which results in a reduced overhead of monitoring, since we can focus on the data we are interested in at the given moment.

Using the OCM-G has several benefits. As an autonomous monitoring infrastructure, the OCM-G provides abstraction and increases modularity — the tools can be developed independently from the monitoring system. The standardization of the communication interface between the OCM-G and tools also gives preconditions for porting the existing tools compliant with the interface to the Grid.

The interface in question is the *On-line Monitoring Interface Specification* (OMIS) [15]. In our experience with OMIS and an OMIS-compliant monitoring system for clusters of workstations, the OCM, we learned that OMIS enables

to adapt existing tools to new programming paradigms, such as PVM or MPI, with minimal changes to the tools' code [2].

3 Monitoring System Interface

Based on our experience in developing an OMIS-based tool environment [2], we decided to use OMIS [15] as a communication interface between the OCM-G and tools. In OMIS, the target system is viewed as a hierarchical set of objects. Currently defined objects are *nodes*, *processes*, *threads*, *messages* and *message queues*. The objects are identified by symbols called *tokens*, e.g. *n_1*, *p_1*, etc.

The functionality of the monitoring system is accessible via *services* which are divided into three classes: *information services* to obtain information about objects, *manipulation services* to manipulate objects, and *event services* to detect events and program actions to be executed whenever a specified event occurs.

The OCM-G will accept all requests compliant with OMIS 2.0. This includes two type of requests: unconditional and conditional. The unconditional requests have a single effect on the target system and yield an immediate response. They are composed of one or more actions which specify either information to be returned or manipulations to be performed. For example, the request *stop process p_1* will attempt to stop the process identified by *p_1*, while the request *return process list for node n_1* will return a list of processes¹ residing on node *n_1*. All requests return a status information (whether the request succeeded or not).

The conditional requests specify an event and a list of actions which shall be executed whenever the event occurs. Thus, responses to conditional requests can be produced multiple times, each time the specified event takes place. Example: *when function X is invoked, return the time stamp of this event*.

OMIS should be extended to fit the Grid requirements and architecture. First, the object hierarchy should be changed to reflect the complex Grid structure. We propose to add a new layer at the highest level of the OMIS object hierarchy — *sites* which are meant to identify administratively uniform domains and encompass multiple systems such as clusters or supercomputers. This extensions will be appropriately reflected in the OCM-G architecture (see 4.2).

Other extensions directly concern the monitoring system's functionality which is reflected in new services. Possible extensions include security-related services, e.g. for authentication, services required for interoperability with the Grid information system or other Grid services [12, 10], and also services required for monitoring new types of events such as generic I/O operations. We also reason that mechanisms already existing in OMIS fit well into the concepts of the Grid Monitoring Architecture described in [18]. For example, the mechanism of event services in fact enables the subscribe operation from consumer to producer which means a request for a continuous stream of data as opposed to a single request/reply.

¹ More exactly: *attached* processes; in OMIS, a tool creates its own context of the target system by explicitly attaching to all objects it wants to deal with.

4 Grid-Enabled OMIS-Compliant Monitor

This Section introduces the OCM-G — a Grid application monitoring system. We start from outlining the requirements imposed on the monitoring system in a Grid environment. Then we present a design of the OCM-G architecture. We conclude with considerations related to the integration of the OCM-G with Grid services.

4.1 Grid-Specific Requirements

The problems we have to deal with when monitoring applications on the Grid are related to reliability, scalability and efficiency of the monitoring system.

Reliability In such a widely distributed system as the Grid, the issue of reliability is twofold: it concerns the problem of keeping information up-to-date and ensuring the mechanism of recovery from crashes, when parts of the monitoring infrastructure are lost, e.g., due to a hardware problem.

Scalability The Grid monitoring system not only will potentially encompass a huge number of systems but it will have to be able to handle a great amount of entities (applications, users, and tools – see section 4.3). Therefore, its design should be properly distributed and decentralized.

Efficiency The problem of efficiency of monitoring is related to the collection and distribution of the monitoring data. The data mostly comes from the detection of events, which may occur excessively, with an unpredictable frequency. To avoid frequent context switches we must report the events in an efficient manner, i.e., provide local buffering instead of reporting each event immediately. On the other hand, in order to not impose high requirements on the local buffer's size, we should provide preprocessing of the monitoring data, i.e. instead of buffering a raw trace of events, store a filtered and/or summarized information. For example, if a measurement concerns a count of certain events, one integer is enough to store this information.

Having the above considerations in mind, we have distinguished three types of components out of which the monitoring system will be composed. They are the *Local Monitor* (LM), the *Service Manager* (SM), and the *Application Monitor* (AM). These components are described in the following section.

4.2 Components of the OCM-G

Parts of the OCM-G are stand-alone components, while other parts reside directly in the application's address space. Fig. 1 shows the monitoring environment. A service manager is the part of the OCM-G to which tools submit requests and from which they receive replies. Service Managers are related to sites in the target system, as shown in Fig. 2. Local monitors reside on each node of the target system. A local monitor receives OMIS requests from a service manager, executes them and passes the reply back to the service manager. An application

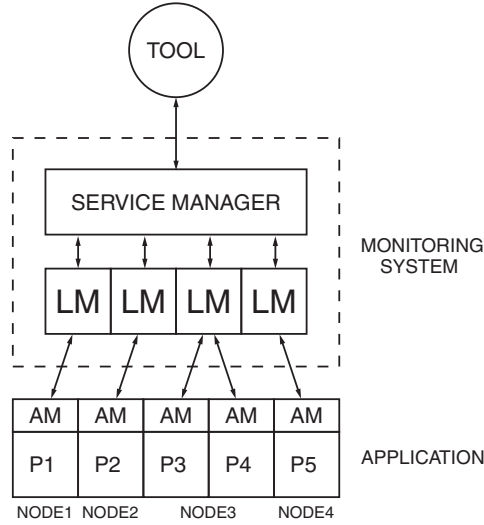


Fig. 1. Architecture of the monitoring environment

monitor is embedded in each process of the monitored application. Its role is to execute performance-critical requests directly in the application context and to buffer performance data before passing it to higher layers.

OCM-G shall interoperate with the Grid information system and use it to manage the startup of the monitoring infrastructure (section 4.3). It might also be enabled to put the monitoring data into the information system if a client demands it. This can be useful for tools that perform statistical observations and need historic data (e.g. 'which files have been most frequently accessed by an application during its many executions?').

Another issue is how the components of the OCM-G should cooperate to efficiently monitor a particular application. The process of assembling the individual components into one monitoring system is described in section 4.3. Here we focus on the logical structure of the OCM-G. As we argue in section 4.3, there should be a single instance of the OCM-G on the Grid which will handle multiple applications. Usually, only part of the monitoring system is involved in monitoring a particular application. This part (i.e. the set of LMs and appropriate SMs) constitutes a *virtual monitoring system* for this application. The components of this virtual system are distinguished by the information about the application they possess. The problem is how the multiple SMs should interoperate to enable the efficient distribution and collection of monitoring requests and data. We propose to designate one of the SMs involved in the application and make it the *Main Service Manager* (MainSM) for the application. The MainSM will hold the collective information about the whole application. This concept is illustrated in Fig. 2. This centralization is useful as it simplifies many algorithms and facilitates keeping the localization information up-to-date. We also hope that it will

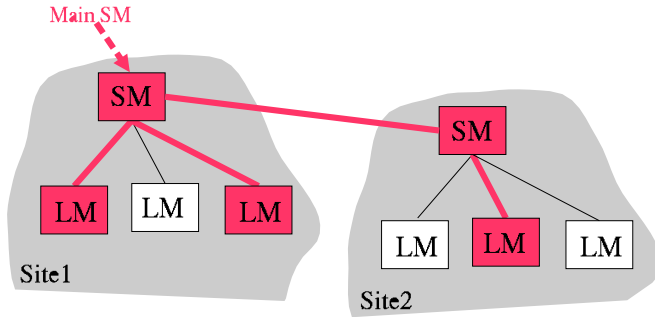


Fig. 2. Virtual monitoring system based on persistent architecture

not put the scalability and efficiency of the system in danger. First of all, the centralization will concern only a single application. Furthermore, the messages will not necessarily be routed through the MainSM. The localization information can also be copied to other SMs and enable the direct communication between them. The special role of the MainSM can be limited to some particular cases, e.g., when a process is migrated, this is first reported to the MainSM which in turn propagates this information to the other SMs.

4.3 Integration with the Grid

The Grid environment is based on many sophisticated technologies that are crucial for achieving the Grid functionality. There are many consequences for the monitoring system due to this fact. In this section, we present some issues concerning cooperation between Grid components and the monitoring system.

OCM-G as a Grid service In the previous OCM versions, each monitored application had its own instance of the monitoring system — the OCM was started just before the application. This scenario seems not suitable for the Grid as a proper start-up of the monitoring system and the application may be problematic. Rather, we should have the monitoring system as a permanently running, global service that will manage multiple applications, users, and tools. With this approach, we can develop a portable mechanism of enrolment of the application at the monitoring system (see below).

Security Some security issues derive from the assumption of having a global monitoring system, since the single monitoring system will handle many users executing different applications. We need an authentication mechanism to ensure that a user is allowed to monitor only his own applications. Users will be identified by means of Grid mechanisms based on the Globus security infrastructure.

OCM-G component discovery Another consequence of the permanent-service approach is that the components of the monitoring system are started

individually, together with the Grid middleware. We need a discovery mechanism to assemble the components into a monitoring system. This is valid both at the initialization stage when the communication between OCM-G components is set up as well as when creating a virtual monitoring system for applications. In the latter case, we must identify those service managers under which the particular application is running and designate the Main SM (section 4.2). The discovery process should be supported by an external mechanism which will be based on the Grid information services.

In the current information systems (such as Globus MDS [12] or R-GMA [7]), the support for global information search is still poor. Moreover, the future of the Grid information systems is still not clear. Thus, we work on a flexible localization mechanism that minimizes the dependence on the information system. We may also use the Grid job scheduler to obtain additional information.

Enrolment of an application at the OCM-G The standard scenario of monitoring an application is that the monitoring infrastructure (be it an autonomous monitoring system or a monitoring module integrated with a debugger) handles the application startup since it is the easiest way to take control over the application. The startup procedure on the Grid has many aspects, among others, resource allocation or copying of input data, and involves many kinds of job schedulers. Therefore, we should aim at not making the process of application startup dependent on the fact of its subsequent monitoring. In other words, the application should be started in a similar way whether it is monitored or not.

The way to achieve that is to make the application self-register in the monitoring system in a bottom-up fashion. With this approach, each application process will contact the nearest LM to register itself in it. The LMs will then pass this information up to the appropriate SMs. The final step is to initialize a connection between all the SMs involved in the application. To do this, we need an external localization mechanism via which the SMs will find themselves. Next, the MainSM for the application will be chosen. In this way, for each application the virtual monitoring system will be created. The only additional requirements posed on the application to be monitored is that it should be linked with additional libraries containing parts of the monitoring system, which are required, among others, for the initial registration in LMs. Some information can also be passed to the application via command line parameters. The application elements will be identified by tokens, which are globally unique symbols, and will be known a priori (before the application start-up) or — if possible — based on unique grid identifiers.

5 Conclusions

We have presented the OCM-G — the Grid-enabled OMIS-compliant Monitoring System aimed for monitoring applications on the Grid. The autonomous nature of the OCM-G and the standardized interface for communication with tools should ease the development of new tools and enable porting, at least in part, of existing tools to the Grid.

The Grid requirements impose a new quality on the monitoring system, which should be properly distributed and decentralized and support efficient data collection and distribution. One of the most important issues is the proper integration of the monitoring system with Grid services.

One of the issues that is still to be addressed is the development of reliability mechanisms to ensure the integrity and proper functioning of the monitoring system in emergency situations, when parts of the Grid hardware or the monitoring system fail. However, this problem is strongly related to the problem of reliable functioning of applications when parts of the Grid fail, which is also a current topic of the Grid research.

References

- [1] Balaton, Z., Kacsuk, P., Podhorszki, N., and Vajda, F.: Comparison of Representative Grid Monitoring Tools. <http://hepunix.rl.ac.uk/edg/wp3/meetings/budapest-jan01/NP%20Grid%20Tools.pdf> 42
- [2] Bubak, M., Funika, W., Baliś, B., and Wismüller, R.: On-line OCM-based Tool Support for Parallel Applications. In: Yuen Chung Kwong (ed.): *Annual Review of Scalable Computing*, 3, Chapter 3, 2001, Singapore 42, 43
- [3] Bubak, M., Funika, W., and Wismüller, R.: The CrossGrid Performance Analysis Tool for Interactive Grid Applications. *Proc. EuroPVM/MPI 2002*, Linz, Sept. 2002 42
- [4] Bubak, M., Funika, W., Żbik, D., van Albada, G. D., Iskra, K., Sloot, P. M. A., Wismüller, R., and Sowa-Pieklo, K.: Performance Measurement, Debugging and Load Balancing for Metacomputing. *Proc. of ISThum 2000 – Research and Development for the Information Society*, pp. 409-418, Poznan, 2000 42
- [5] Bubak, M., Funika, W., Baliś, B., and Wismüller, R.: A Concept For Grid Application Monitoring. In: *Proc. PPAM 2001*, Nałęczów, Poland, September 2001, pp. 307-314, LNCS 2328, Springer 2002 42
- [6] CrossGrid Project: <http://www.eu-crossgrid.org> 41, 42
- [7] DataGrid Project: <http://www.eu-datagrid.org> 42, 47
- [8] Foster, I., Kesselman, C. (eds.): *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999 41
- [9] Foster, I., Kesselman, C., Tuecke, S. The Anatomy of the Grid. *International Journal of High Performance Computing Applications* 15 (3) 200-222 (2001); <http://www.globus.org/research/papers/anatomy.pdf> 41
- [10] Foster, I., Kesselman, C., Nick, J. M., and Tuecke, S.: The Physiology of the Grid. January 2002, http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf 41, 43
- [11] Gerndt, M., Esser, H.-G.: Specification Techniques for Automatic Performance Analysis Tools. *Proc. 8th Int. Workshop on Compilers for Parallel Computers CPC 2000*, Ecole Normale Supérieure Lyon, January 2000, Aussois, pp. 1-11 42
- [12] Globus Project: <http://www.globus.org> 43, 47
- [13] Gunter, D. et al.: NetLogger: A Toolkit for Distributed System Performance Analysis. *Proc. IEEE Mascots 2000 Conference*, Lawrence Berkeley Natl. Lab., Report LBNL-46269 42
- [14] Heartbeat Monitor Specification. http://www.globus.org/hbm/heartbeat_spec.html 42

- [15] Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997) <http://www.bode.in.tum.de/~omis> 42, 43
- [16] Podhorski, N., Kacsuk, P.: Design and Implementation of a Distributed Monitor for Semi-on-line Monitoring of VisualMP Applications. Proc. DAPSYS 2000, Balatonfured, Hungary, 23-32, 2000 42
- [17] Shaffer, E. et al.: Virtue: Immersive Performance Visualization of Parallel and Distributed Applications. IEEE Computer, 44-51 (Dec. 1999) 42
- [18] Tierney, B., Aydt, R., Gunter, D., Smith, W., Taylor, V., Wolski, R., Swany, M., et al.: White Paper: A Grid Monitoring Service Architecture (DRAFT), Global Grid Forum. 2001 <http://www.didc.lbl.gov/GridPerf> 41, 43
- [19] Vetter, J. S., and Reed, D. A.: Real-time Monitoring, Adaptive Control and Interactive Steering of Computational Grids. The International Journal of High Performance Computing Applications 14 357-366 (2000) 42
- [20] Waheed, A., et al.: An Infrastructure for Monitoring and Management in Computational Grids. Proc. 5th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, (March 2000) 42
- [21] Wolski, R., Spring, N., and Hayes, J. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Future Generation Computer Systems 15 757-768 (1999) 42

The CrossGrid Performance Analysis Tool for Interactive Grid Applications^{*}

Marian Bubak^{1,2}, Włodzimierz Funika¹, and Roland Wismüller^{3,4}

¹ Institute of Computer Science, AGH
al. Mickiewicza 30, 30-059 Kraków, Poland

² Academic Computer Centre – CYFRONET
Nawojki 11, 30-950 Kraków, Poland

³ LRR-TUM, Inst. f. Informatik, Technische Universität München
D-80290 München, Germany

⁴ Institute for Software Sciences, University of Vienna
A-1090 Vienna, Austria

Abstract. The EU CrossGrid project aims at widening the use of Grid technology for interactive applications. In order to meet the efficiency constraints of these applications, a Grid applications performance analysis tool is developed that not only allows the standard measurements, but also supports application-specific metrics and other high-level measurements. The article outlines the projected functionality of this tool, as well as design considerations and a use case.

1 Introduction

Grid computing has the potential of providing a cost- and resource-efficient way of solving data and compute intensive problems. In the EU DataGrid project, for instance, Grid technologies are used to enable a world wide, efficient access to the huge amounts of data that will be produced by future High Energy Physics (HEP) experiments. In March 2002, a new EU project – CrossGrid [4] – was launched to extend existing Grid technologies towards a new direction: *interactive applications*. Besides providing the necessary Grid services and test-beds, four demonstrators for interactive Grid applications are being developed in CrossGrid [3]: simulation of vascular blood flow, flooding crisis team support tools, distributed data analysis in HEP, and simulation of air pollution combined with weather forecast.

Efficiently using the Grid as an environment for such large-scale interactive applications requires programmers to exactly know the performance behavior of their applications. However, even with a good knowledge of an application's code, its detailed run-time behavior in a Grid environment is often hard to figure out, not the least because of the dynamic nature of this infrastructure. In order to support this task, the CrossGrid project develops a tool, named G-PM, which not only allows to measure just the standard performance metrics, but allows to

^{*} Partly funded by the European Commission, project IST-2001-32243, CrossGrid

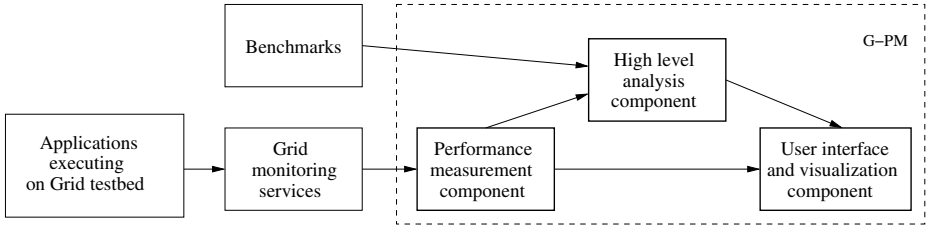


Fig. 1. Component structure of G-PM

determine higher-level performance properties and application specific metrics. E.g., for an interactive client/server application it is important to measure the response time between a user interaction and the proper reaction of the program. Even more important is the breakdown of this time, e.g. how much of it is due to the client's computation, due to network delays, or due to the server's computation.

For providing this kind of information, the G-PM tool uses three sources of data: performance measurement data related to the running application, measured performance data on the execution environment, and results of micro-benchmarks, providing reference values for the performance of the execution environment. This acquired data is heavily processed and transformed within G-PM in order to take into account the structural and architectural details of the Grid. In this way, information on performance properties and/or bottlenecks in applications can be provided without a need for the user to know the Grid's detailed architecture.

The paper is organized as follows: Section 2 outlines the projected functionality of the tool, Section 3 presents some details on its design, while Section 4 contains a simple use case. We conclude with a discussion of related work and the current status of our own work.

2 Projected Functionality of G-PM

The performance analysis tool G-PM is structured into three major components (c.f. Fig. 1) that enable a user to optimize the performance of a Grid application:

1. performance measurement component (PMC),
2. component for high level analysis (HLAC),
3. user interface and visualization component (UIVC).

The PMC provides the functionality for basic performance measurements of both Grid applications and the Grid environment. The results of these basic measurements can be directly visualized by the visualization component. In addition, they can serve as an input to the high level analysis component.

The HLAC supports an application and problem specific analysis of the performance data. On the one hand, it allows to measure application specific performance metrics. On the other hand, it provides a specification language which

allows to combine and correlate different performance measurements in order to derive higher-level metrics, like e.g. load imbalance. The objective of this approach is to provide more meaningful data to application developers.

Finally, the UIVC allows the user to specify performance measurements and visualizes the performance data produced by PMC and/or HLAC.

In the following subsections, we provide more details on these components.

2.1 Performance Measurement Component

The PMC supports two types of measurement requests: *Application related requests* ask for information on the whole or a part of a Grid application. They can be issued either before the application is started, or while it is already running. *Infrastructure related requests* ask for information on the whole or a part of the Grid infrastructure.

The PMC can acquire the following types of application related performance data:

1. *data transfer*: amount of data received from / sent to other processes / hosts, amount of data read from / written to files,
2. *resources utilization*: CPU usage (computation time), memory accesses (number, frequency, time), disk accesses, cache operations,
3. *delays*: due to communication, synchronization, and I/O operations.

The Grid infrastructure related performance data includes:

1. *availability of resources*,
2. *dynamic resource information*: CPU load and link load,
3. *static resource information*: CPU architecture, CPU performance, memory hierarchy features, memory performance, and secondary storage features of nodes, bandwidth and latency of links.

Measurements can be defined with different granularities in space and time. In space, it should be possible to apportion the measurements with respect to sites and hosts, and – for application related requests – also with respect to processes and functions inside a process. In time, three different granularities are provided:

1. single performance value comprising a complete execution,
2. summary values for a selectable period of time,
3. progression of a performance metrics over time, with selectable resolution.

The PMC can be used directly via the UIVC. For example, let us assume that the user requests the measurement of the total volume of communication between two processes. To define this measurement, the user selects the “*received data volume*” and the “*sent data volume*” metrics, the process identifiers (resolution in space), and the interval of the measurement (resolution in time). In a visualization window, G-PM then displays the requested performance information in one of several possible forms, e.g. as two bar-graphs, one for each direction of message transfer, or e.g. as a communication matrix where a pair (column, row) represents a *sending process* – *receiving process* pair with corresponding communication volume values.

2.2 High Level Analysis Component

The HLAC provides two major functions to the user:

1. It enables to combine and/or correlate performance measurement data from different sources. E.g. it allows to measure load imbalance by comparing an application's CPU usage on each node used. In a similar way, the portion of the maximum network bandwidth obtained by an application can be computed by comparing performance measurement data with benchmark data.
2. It allows to measure application specific performance properties, e.g. the time used by one iteration of a solver, the response time of a specific request, convergence rates, etc.

To achieve the latter, the application programmer needs to mark relevant places in the code, like e.g. the beginning and the end of a computation phase. This is done by inserting a special function call (*probe*) into the application's source code. Any number of scalar parameters may be passed to a probe, which allows to provide application-specific data relevant for performance analysis. Since performance is so crucial for interactive Grid applications, we expect that programmers are willing to insert this small and limited amount of instrumentation into their applications in order to get more adequate performance information. Once the probes are available in an application, G-PM allows to measure:

- any application-related or infrastructure-related metrics mentioned in subsection 2.1 in an execution phase defined by any two probe executions,
- application-specific metrics defined by any parameter passed to any probe,
- any metrics that can be (recursively) derived from already existing ones.

The measured values can be accumulated over the whole run-time of the application, plotted as a graph against real time, or against the number of executions of specified probes (i.e. against the application's execution phases).

The specification language ASL, developed in the IST Working Group APART [5], is used to define application-specific metrics, i.e. to specify, how these metrics are computed from the data delivered by the application's probes and the generic performance measurement data provided by the PMC. Thus, using the same probes, different metrics can be measured, e.g. minimum, maximum and mean time for an iteration, amount of communication per iteration, etc. As mentioned, probes may also pass application-specific data to the HLAC, e.g. a residuum value that allows to compute a solver's convergence rate.

For each application, the ASL specifications of all application-specific performance metrics can be stored in a configuration file. The user of G-PM can load the proper file at run-time to customize the tool for his application. Once the specification file is loaded, all defined metrics appear in the UIVC's measurement definition window and can be selected in the same way as the standard metrics. Note that the probes in the application code themselves do not define any measurement, they just mark interesting points in the execution. An actual measurement is performed only when requested by the user via the UIVC. The overhead of an inactive probe, which is a call to a function containing a

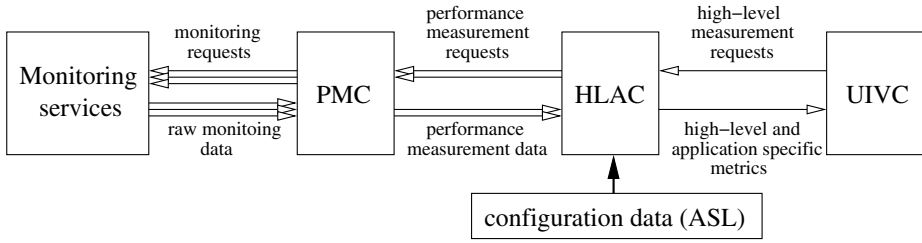


Fig. 2. Request and data transformation in G-PM

non-executed conditional block, is negligible in all practical cases. For a probe receiving a `double` value as an argument, we measured $79ns$ on a 300 MHz Sun UltraSPARC-II.

In addition to application-specific ASL files, a generic ASL file bundled with G-PM contains the definitions of higher-level metrics independent of a specific application. Examples of such metrics are load imbalance and relative CPU/network performance. The last two metrics present an application's performance as a percentage of the execution platform's "optimal" performance, which is measured via micro-benchmarks.

2.3 Graphical User Interface Component

The G-PM tool provides a graphical user interface based on the PATOP performance analysis tool [2]. It consists of four different classes of windows:

1. The main window offers a menu to enter user commands and displays a list of all active performance measurements, as well as a list of all open visualization windows. These lists are used to manage (i.e. modify or delete) both measurements and visualization windows.
2. A measurement definition window to specify the performance measurements to be performed (type of metrics, objects involved, time interval, location of the measurement).
3. A visualization definition window. It allows to specify the way in which a selected set of performance measurements should be visualized. These different ways include bar graphs as well as graphs plotted against real time or against the number of executions of specified probes.
4. An arbitrary number of visualization windows.

3 Design Considerations

Besides providing the user interface for defining and visualizing performance measurements, the main task of G-PM is to transform the user's (high-level)

measurement requests into low-level requests that can be handled by Grid monitoring services. Vice versa, G-PM has to compute performance metrics from the raw measurement data returned by the monitoring services (c.f. Fig 2).

In the following, we will focus on these transformations within the PMC and the HLAC, as well as the requirements imposed on the Grid monitoring services.

3.1 Grid Monitoring Services

G-PM is based on the premise that there exist Grid monitoring services that provide on-line, low-level data on the Grid infrastructure and running applications. Typical services will e.g. return the current CPU load of a node or monitor the occurrence of events in an application process. The PMC uses these services to implement performance measurements. For example, consider the measurement of the total volume of communication between two processes mentioned in subsection 2.1. The PMC asks the monitoring services to capture all “*message sent*” events (i.e. invocations of the communication library’s send function) between these processes, and to add the message’s size to a previously defined counter.

The actual interface between the PMC and the monitoring services is based on the OMIS 2.0 specification [10] and is adapted and further developed within the CrossGrid project. For a more detailed discussion, see [1]. The interface accepts monitoring requests, which may belong to one of the following types:

1. *information requests* just return some information on the infrastructure or application in the reply,
2. *manipulation requests* result in a manipulating action,
3. *event requests* capture a specified event and trigger a list of actions (either for returning information or for manipulation).

The G-PM tool mainly uses information requests and event requests, e.g.:

- The information request *return CPU load on nodes n_1 , n_2 , n_3* immediately returns a list of three numbers containing the load values.
- To perform measurements related to a function A within an application, event requests with events: “*function A has started*” and “*function A has ended*” are used. Whenever these events occur, a list of actions is executed by the monitoring services.

These actions then implement the actual performance measurements, e.g. to obtain the execution time or the volume of data transferred. Among others, the following actions are available for this purpose:

- *increment a named counter or integrating counter* by a specified amount (which may be defined by a parameter describing the current event),
- *store a record* (which may include parameters of the current event) in an event trace,
- *send event information* immediately to the PMC,
- *enable* or *disable* other event requests (which results in starting or stopping other performance measurements).

To support the HLAC, the monitoring services offer two additional features: First, they allow to monitor probes in the application's code. This is similar to the “*function A has started*” event mentioned above. Second, they allow to access the results of micro-benchmarks: given a host and a host-related metrics, return the result for the specified host; given two hosts and a network-related metrics, return the result for the network connection between these hosts.

3.2 Performance Measurement Component

The PMC supplies performance measurement data on applications and the Grid infrastructure on request from the UIVC or the HLAC. Such a request may include the *metrics to be measured* (time, data volume), an *object(s) specification* (site, host, process), a *time interval* (complete execution, time/program phases), a *location* (program, module, function), and *constraints* (metrics limit values, communication partners).

The basic task of the PMC is to implement these requests by means of the available monitoring services. For example, when a measurement “*execution time of function A in processes p_1, p_2, p_3*” is requested, the PMC ensures that:

- the measurement is transformed into two event requests (“*function A has started / ended*”) for each process, with appropriate actions, which use an integrating counter to compute the difference of the end time and the start time of function *A* in each of the processes,
- these requests are submitted to the monitoring services,
- the data returned by the monitoring services is processed and returned in the required form to the requesting component (UIVC or HLAC).

3.3 High Level Analysis Component

While the PMC transforms raw monitoring data to performance measurement data, the HLAC further transforms the latter to higher-level and/or application specific properties (see Fig. 2). The main differences between these two components are:

- While the transformations in the PMC are fixed, the HLAC is fully configurable with respect to the performance measurement data it gets as input, the high-level properties it should provide to the user, and the way how these properties are computed from the different types of performance measurement data.
- Since the HLAC is configurable, it can use / provide application specific performance data. This can be achieved by loading application specific configurations on demand of the user.

Similar to the PMC, the HLAC must provide transformations in two directions: When the user requests some high-level property to be measured, the HLAC requests the PMC to measure all the required performance measurement

data. Vice versa, when this data arrives, the high-level properties are computed according to the rules specified in the HLAC's configuration data.

The configuration data is based on the APART Specification Language ASL, which allows to specify both a data model for the incoming performance measurement data, as well as computation rules for the performance properties. Thus, the HLAC basically implements an interpreter for the ASL.

4 A Use Case

The medical application developed in the CrossGrid project will make use of a special library, called Grid Visualization Kernel (GVK) [8] for the communication between its simulation and visualization components. Assume that the programmer wants to analyze the performance of the visualization, in particular the impact of GVK. In the following, *server* denotes a host executing parts of the simulation, while *client* is the host visualizing the results.

1. In order to acquire GVK specific performance data, the programmer (manually) inserts probes at three places in the source code of GVK:
 - (a) immediately after (on the server) GVK is requested to visualize a frame,
 - (b) immediately after (on the server) the data is sent to the client,
 - (c) immediately before the data is passed to the graphics engine on the client. The size of the data is passed to the probe as an argument.
 After inserting the probes, GVK needs to be recompiled.
2. The programmer writes an ASL specification for new metrics related to the GVK, for instance:
 - Generated image frames per second = $1 / (\text{Time between successive invocations of probe } a)$.
 - Compression factor = $(\text{data size passed to probe } c) / (\text{communication volume from server to client between executions of probe } a \text{ and } b)$. The communication volume between hosts is a standard metrics of the PMC.
 - GVK processing time for a frame = time interval between the execution of probe *a* and the execution of probe *b*.

Note that the same probes are used to define different metrics. Since these probes and metrics are of more general interest, they may already have been inserted / defined by the developer of GVK. In this case, the GVK library will already come with an accompanying ASL specification file for G-PM.

3. The programmer links his application with the instrumented GVK library from step 1, as well as with an instrumented communication library. The latter one is part of the G-PM distribution.
4. The programmer starts his application under control of G-PM.
5. He loads the GVK-specific ASL file into G-PM. This is the file created in step 2. After the file has been loaded, the measurement definition dialog of the UIVC shows the GVK-specific metric in addition to the standard metrics.
6. The programmer can now select a measurement for “*generated image frames per second*” in the measurement definition window. In addition, he can select a suitable visualization style (e.g. as a function over time).

7. If the programmer now wants to measure the communication volume within the simulation component for each generated image frame, he can define the new metrics on-line, without a need to change or even restart the application, since the probe required for this metrics is already available.

5 Related Work

There already exists a couple of performance tools for the Grid. In contrast to G-PM, most tools that support the monitoring of applications are based on an off-line analysis of event traces. A prominent example is NetLogger [7]. Besides the problem of scalability, trace-based off-line tools are not useful for the analysis of interactive applications, since they cannot show the performance metrics concurrently with (i.e.: in correlation to) the user's interaction with the application. Currently, on-line tools are available only for infrastructure monitoring, since their main use is for resource management, which requires on-line data. Examples are the Heartbeat Monitor [6] and the Network Weather Service [14] that forecasts network and node performance.

Some performance tools implement ideas closely related to the ones presented in this paper. User-defined instrumentation, in particular timers that allow to measure the CPU time of code sections, is used in the TAU performance analysis environment [12]. Autopilot [13], a distributed performance measurement and resource control system, has a concept called *sensors* that very closely corresponds to our probes. In both of these systems, however, the metrics derived from this instrumentation are fixed by the tool's programmer and cannot be configured by the tool's user. A metrics definition language (called MDL) is used in the Paradyn tool [11]. However, the MDL [9] is hidden inside the tool and is not intended for the definition of application-specific metrics. Paradyn is one of the few performance tools that strictly follow the on-line approach, where even the instrumentation is inserted and/or activated during run-time. Another such tool is PATOP [2], which forms a basis of G-PM.

The main contribution of G-PM is its unique combination of Grid awareness, on-line measurement, and automatic instrumentation for standard metrics on the one hand with a support for manual instrumentation and user-definable metrics on the other.

6 Status and Conclusions

We have recently finished the requirements specification and the design phase of the G-PM tool. The first prototype of G-PM is currently being implemented and will be available in February 2003. This prototype will include basic performance measurements, and some examples of higher-level metrics, but will not yet be fully configurable via the ASL. This feature will be added in the next prototype available in 2004. The final version of G-PM will be ready by the end of 2004. Like all other system software developed in CrossGrid, G-PM will then be freely available via a public software license.

We expect that the highly pre-processed, application-specific performance information provided by G-PM will significantly simplify program optimization and tuning, making the work of programmers and scientists more efficient, and thus eliminating possible barriers to work on the Grid.

References

- [1] B. Baliś, M. Bubak, W. Funika, T. Szeplieniec, and R. Wismüller. An Infrastructure for Grid Application Monitoring. In *Proc. 9th European PVM/MPI Users' Group Meeting*, Linz, Austria, Sept. 2002. LNCS, Springer-Verlag. 55
- [2] M. Bubak, W. Funika, B. Baliś, and R. Wismüller. On-Line OCM-Based Tool Support for Parallel Applications. In Y.C. Kwong, editor, *Annual Review of Scalable Computing*, volume 3, chapter 2, pages 32–62. World Scientific Publishing Co. and Singapore University Press, 2001. 54, 58
- [3] M. Bubak, M. Malawski, and K. Zajac. Towards the CrossGrid Architecture. In *Proc. 9th European PVM/MPI Users' Group Meeting*, Linz, Austria, Sept. 2002. LNCS, Springer-Verlag. 50
- [4] CrossGrid Project. <http://www.eu-crossgrid.org>. 50
- [5] T. Fahringer, M. Gerndt, G. Riley, and J.L. Träff. Knowledge Specification for Automatic Performance Analysis – APART Technical Report. Technical report, ESPRIT IV Working Group on Automatic Performance Analysis, Nov. 1999. <http://www.par.univie.ac.at/~tf/apart/wp2/report-wp2.ps.gz>. 53
- [6] The Globus Heartbeat Monitor Specification. <http://www-fp.globus.org/hbm/heartbeat-spec.html>. 58
- [7] D. Gunter et al. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proc. IEEE Mascots 2000 Conference*, Aug. 2000. Lawrence Berkeley Natl. Lab., Report LBNL-46269. 58
- [8] P. Heinzlreiter, D. Kranzlmler, G. Kurka, and J. Volkert. Interactive Visualization in Large-Scale, Distributed Computing Infrastructures with GVK. In *SCI 2002, 6th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Florida, July 2002. 57
- [9] J.R. Hollingsworth et al. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, CA, USA, Nov. 1997. ftp://grilled.cs.wisc.edu/technical_papers/mdl.ps.gz. 58
- [10] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker-Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7. <http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz>. 55
- [11] B.P. Miller et al. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, Nov. 1995. <http://www.cs.wisc.edu/paradyn/papers/overview.ps.gz>. 58
- [12] S. Shende, A.D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proc. ACM SIGMETRICS Symp. on Parallel and Distributed Tools (SPDT '98)*, pages 34–145, Welches, Oregon, Aug. 1998. 58
- [13] J. Vetter and D. Reed. Real-time Monitoring, Adaptive Control and Interactive Steering of Computational Grids. *The International Journal of High Performance Computing Applications*, 14:357–366, 2000. 58

- [14] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15:757–768, 1999. 58

Current Trends in Numerical Simulation for Parallel Engineering Environments

Carsten Trinitis and Martin Schulz

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Institut für Informatik, SAB, Technische Universität München
80290 München, Germany
{Carsten.Trinitis,Martin.Schulz}@in.tum.de
<http://wwwbode.in.tum.de/>

For the first time, the program of the EuroPVM/MPI conference series includes a special session on current trends in Numerical Simulations for Parallel Engineering Environments. Its goal is to provide a discussion forum for scientists from both the engineering disciplines and computer science in order to foster a closer cooperation between them. In contrast to traditional conferences, emphasis is put on the presentation of up-to-date results with a short turn-around time, which could also include the presentation of work-in-progress.

We have selected four papers from various backgrounds to be presented in this special session. They cover both specific simulation applications and general frameworks for the efficient execution of parallel simulations. This selection thereby represents the intended mix between specific problems and requirements given with particular applications or environments and general solutions to cope with them. We think that this resulted in an attractive program and we hope that this session will be an informal setting for lively discussions.

Several people contributed to this event. Thanks go to Jack Dongarra, the EuroPVM/MPI general chair, and to Peter Kacsuk, Dieter Kranzlmüller, and Jens Volkert, the PC chairs, for giving us the opportunity to establish this special session. Dieter Kranzlmüller also deserves additional credit for his support in including the contributions for this special session in the EuroPVM/MPI proceedings and for the numerous discussions and his encouragement, which got the work on this special session started. Last but not least, we would also like to thank the numerous reviewers and all those who took the time to submit papers to this new event.

We hope this session will fulfill its purpose to encourage interdisciplinary cooperation between computer scientists and engineers in the various application disciplines and that it will develop into a long and successful tradition in the context of EuroPVM/MPI.

Automatic Runtime Load Balancing of Dedicated Applications in Heterogeneous Environments

Siegfried Höfinger

Department of Theoretical Chemistry and Molecular Structural Biology
Molecular Dynamics and Biomolecular Simulation Group
University of Vienna, Währingerstr. 17, A-1090, Vienna, Austria
sh@mdy.univie.ac.at
<http://www.mdy.univie.ac.at>

Abstract. A new and fully-automatic version of dynamic Load Balancing on a Wide Area Network cluster consisting of 5 UNIX-type machines of different architectures and different operating systems is presented based on the recently described method of dynamic Speed Weighted Load Balancing in the context of Quantum Chemistry calculations with GREMLIN. The cluster itself is set up via PVM 3.4.3 and the 5 machines are located in some research centres in France, Italy and Austria and for internode communication we only allow the secure shell protocol. The automatic character is realized by repeated calls to a special dedicated load balancing tool that takes into account differing individual node performances via Speed Factors, which are altered periodically depending on the current cluster conditions.

Keywords: WAN-cluster, Load Balancing, Quantum Chemistry, PVM, Heterogeneous Cluster

1 Introduction

Cluster architectures are becoming the standard platform in current High Performance Computing [1]. Its increasing popularity and usage may enable the existing boundaries of computation to be extended in all the related disciplines in science and engineering. On the other hand the increasing demand for high-end computational solutions also puts some pressure on the software development site. This is because most of the applications at first have to be adapted to really be capable of making use of powerful hardware and parallel architectures. This observation is particularly important when it comes to parallel performance in heterogeneous environments, where the "parallel machine" is constructed from an adequate collection of ordinary computers (of usually entirely different type and different individual performance) whereby all these various machines are usually interconnected by ordinary low-bandwidth Internet connections and may be physically located far apart from each other. In terms of software this not only means having to port existing code onto parallel architectures, but also to

account for the heterogeneity of the cluster during distribution of the parallel work. Nevertheless, scalability of the parallel code is still the only overall limiting criterion for any parallel application, since it does not make sense to run a computational problem on a huge parallel cluster, where most of the nodes do not actually perform calculations but mainly idle and wait for another computational bottleneck to terminate.

Science and engineering may be considered the major recipients of all the high performance computing efforts. Typically the nature of a given problem is such that it is feasible to achieve scalable parallel performance. This is because many of these applications very often exhibit a predictable and very sharp profile in the relative CPU intensity of all the occurring subroutines and thus spend almost all of the CPU-time in a few significant modules. In addition, many typical problems in science and technology are solved in cycles, where the same procedure is repeated over and over again until either a certain accuracy threshold is surpassed, or a predefined overall number of cycles is completed. Parallel programs may profit from this cyclic periodicity by applying changes to the way the parallel work is distributed in between two such subsequent cycles. In this sense, dynamic run-time regulation and dynamic load balancing may be realized and varying cluster conditions may be noticed in an early stage and taken into account accordingly.

Finally, an ideal scenario would be, if the participating nodes (forming a cluster on which some application is run in parallel) were able to identify any changes in cluster conditions themselves and answer accordingly in an entirely autonomous fashion. In a previous communication it was shown how a typical quantum chemical application — GREMLIN [2] — could actually benefit from such a dynamic run-time interference considerably [5]. However, with this particular example the actual process of interfering with the running program took place manually and needed careful monitoring of the run-time behavior of all of the participating hosts. Therefore in the present report an enhancement to the already promising approach shall be described and a demonstration given of how dynamic cluster event handling may be maintained in an fully-automatic self-regulatory manner.

In the next section a problem description is given and the underlying load balancing technique described. Section 2 gives details of the WAN cluster set-up. In section 3 the behaviour of a real-world example is analyzed and finally conclusions are drawn in section 4.

1.1 Problem Description

Although not generally limited to Quantum Chemistry alone, we shall for the moment concentrate on only this kind of application that shall be run on a Wide Area Network (WAN) cluster of computers. Quantum chemical calculations exhibit an almost perfect pattern of characteristic attributes that make them particularly interesting targets for a parallel approach.

- First of all, from the point of view of computational cost they are extremely expensive and usually the CPU demand for medium to large-scale problems makes their routine use prohibitively expensive.
- Second, because large-scale problems cannot be handled from memory-intensive variants, the community has long been trained in moving the computational load from memory to CPU-intensive processing. As a direct result of this nowadays only so-called direct methods are used and these direct algorithms are usually much better suited to parallelization.
- Quantum Chemical calculations show a very narrow profile of how the total CPU time is spent on different parts of the code. This means that most of the computational time goes into only a few subroutines, which therefore become the targets of all parallel operations.
- Quantum Chemical problems are solved in iterative cycles (SCF) (typically 5 to 20 cycles depending on the quality of the initial guess wave function). All these individual cycles have large independent execution times. Therefore if one such cycle can be properly parallelized then there is a good chance that scalable parallel performance may be achieved with little communication overhead and pretty long lasting individual parallel intervals.
- Cyclic repetitions of one process are often a characteristic feature of many large-scale scientific and engineering applications. One can easily think of a dynamic regulation and load equilibration mechanism which should become active at the beginning of each of these cycles and react upon the cluster conditions observed in the previous, hence just finished, cycle. In this sense the permanent adaption to current actual run-time conditions of all the participating cluster machines would certainly assist in making the whole process more flexible. This in turn would improve parallel scalability significantly making it possible to enlarge the cluster further for example by adding another group of machines enabling a further speed-up.

There are, however, also drawbacks and problems to a straightforward implementation of parallel Quantum Chemical calculations. To begin with, most of the CPU time is spent on the calculation of ERIs, the Electron Repulsion Integrals, which are 6-dimensional, 4-center integrals over the basis functions.

There is a vast literature on how actually ERIs may be best computed. The very method selected here originates from Obara and Saika [3]. The problem with all these ERI methods is that they are complex recursive algorithms, where a certain set of ERIs must be computed altogether because all the individual items depend on each other. The relative individual cost of these ERI blocks can be rather different and may change dramatically from one block to another. This coarse-grained structure obviously has to be taken into account when distributing the net ERI work onto a group of parallel processing machines and this particular problem has already been addressed by a special dedicated partitioning tool [2]. Further complications arise when dealing with machines of different quality. In such a case it will be important to consider the varying stand-alone performance of all the cluster forming computers, which may be managed by introducing machine specific Speed Factors and employing SWLB (Speed Weighted Load



Fig. 1. Symbolization of the block structure in recursive ERI computation. The principal situation (left) as opposed to parallel runs in homogeneous (middle) and heterogeneous (right) environments

Balancing) [4]. This would just alter the way of distributing different ERI blocks onto different machines.

If the overall number of ERIs for a given problem was symbolized by the area of a square, then the typical block structure could be represented from a schematic picture like the one shown in Fig. 1 (leftmost panel). Partial areas mark off blocks of integrals that have to be computed together because the recursive relations relate them to one another. Moving the whole problem onto a parallel machine, with equally fast performing CPUs, we could think of a possible representation as shown in the middle panel of Fig. 1. This picture would change again if we move forward to a heterogeneous cluster with CPUs of different speeds (rightmost panel of Fig. 1). An additional problem arises if in the course of a running calculation some of the cluster machines undergo crucial changes in terms of CPU workload. For example, when unforeseen third party processes suddenly become active on a certain machine, then the usual fractional load of this affected host must be lowered, because the additional job, though not serving the Quantum Chemistry calculation, will also consume CPU-time. Therefore a dynamic interference to the running program must be enabled in order to allow repartitioning of the ERI work according to the changed cluster situation. This has already been considered in the dynamic variant of Speed Weighted Load Balancing [5]. This version however needs further refinement, because it is based on the manual modification of the Speed Factors of the machines throughout the runtime of an active job. This is possible for a small cluster and careful load monitoring, but cannot be accomplished in a more general situation, where one wants to have a more automatic solution.

1.2 Computational Implementation

The enhancements to the recently described version of dynamic SWLB [5] are twofold. First of all, in the context of PVM (Parallel Virtual Machine, rel.3.4.3) [6] one needs to introduce direct 1:1 control over parallel tasks of different machines at different sites in the master part of the program code, especially if it comes to spawning more than one task on one particular host of the WAN cluster (e.g. on a parallel machine serving for several cluster nodes). If the different

nodes are not addressed in a controlled way from the master host, then PVM will take over and apply its linear scheduling policy. This is then much harder to oversee. However if this is done program-internally at the master host, then one always has a much more clearly defined influence on all the Speed Factors associated with all the parallel working hosts. Therefore we define a WAN cluster configuration file that lists all the participating hosts with their full IP name (multiple entries if there are several CPUs of a parallel machine) together with their corresponding actual Speed Factor. It is this file (CLSPFK) that is read and updated periodically when the load is distributed over the nodes.

Secondly we also record the actual consumed CPU-time of each parallel task in each of the cyclic iterations. Technically speaking we make use of the quasi-standard `TIME()` function under FORTRAN 77. Having obtained all the individual node times, we compute an average node interval and scale all the current Speed Factors by the fraction $\frac{\text{average node time}}{\text{actual node time}_I}$. The scaled and thus adapted Speed Factors are written back to the CLSPFK file and the partitioning tool is re-invoked to work on the newly defined Speed Factors found in CLSPFK. The likewise determined portions of ERI work are distributed over the nodes. In this way we achieve an entirely self-defined cluster regulation and shall demonstrate its usefulness in the following sections.

2 WAN Cluster Description

The WAN-cluster was set up with five machines located at *GUP* LINZ (A) (2 machines), *RIST*⁺⁺ SALZBURG (A) (1 machine), *ICPS* STRASBOURG (F) (1 machine) and *G. Ciamician* BOLOGNA (I) (1 machine). Internode communication was realized using PVM (Parallel Virtual Machine, rel.3.4.3) [6] based on 1024 bit RSA authenticated ssh-connections. Individual benchmark runs on the small test system glycine/631g (10 centre, 55 basis functions, result:-271.1538 Hartree, 17 iterations) revealed initial WAN-cluster conditions as shown in Table 1. Speed factors were derived from the glycine-benchmark and reflect performance ratios with respect to the slowest machine (node I). Node II was equipped with two different kinds of CPU which are offered arbitrarily to the user. Therefore the Speed factor for node II in Table 1 is due to the CPU of R10000 type and the "+" shall imply that this factor will immediately increase all the time the user gets access to the faster CPUs of R12000 type. The network bandwidth data shown in the second last column was obtained from measuring transfer rates between nodes and the future master-machine (node III).

3 Discussion

In Fig. 2 we show the results of the Hartree-Fock (HF) calculation [7] [8] with GREMLIN [2] of the molecular system alanine at aug-cc-pVDZ basis set quality (13 atoms, 213 basis functions, S,P,D types, 4004 primitive gaussians, result:-321.7924 Hartree) when employing dynamic SWLB with auto-regulation.

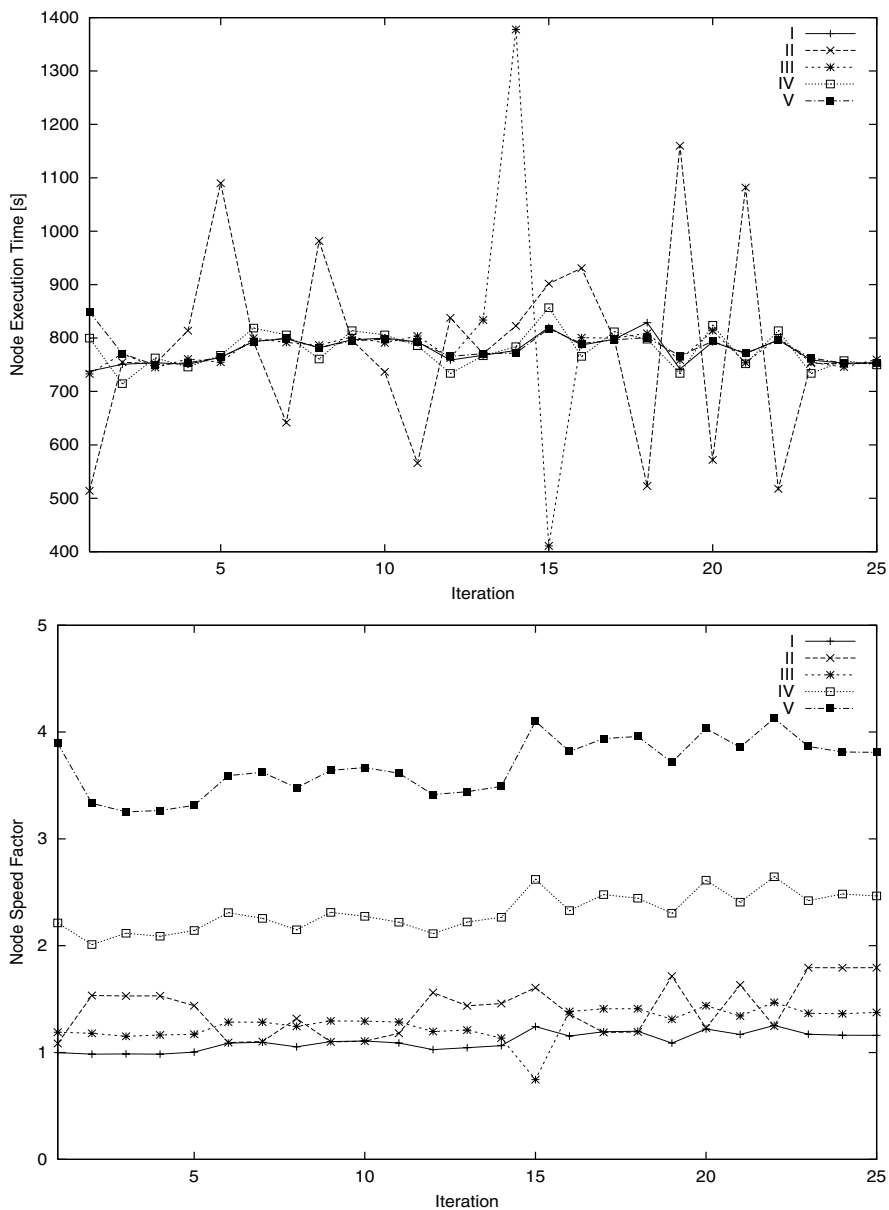


Fig. 2. 5-node runtime profile of a typical parallel SCF calculation (Hartree Fock) with the Quantum Chemistry program GREMLIN employing auto-regulated dynamic Speed Weighted Load Balancing

Table 1. Speed-Factor and Network-Latency table for the WAN-cluster

| Physical Location | Architecture/ Clock Speed/ RAM/2L-Cache | Operating System | Relative Speed Factor | Network Bandwidth [kB/s] | Exp.Total Comm. Time [s] |
|---|---|------------------|--------------------------|-----------------------------|--------------------------------|
| node I G.C. BOLOGNA Italy | INTEL P II 333 MHz 256 MB/512 KB | LINUX 2.2.14 | 1.000 | 166 | 128 |
| node II ICPS STRASBOURG France | MIPS R10000/R12000 200 MHz/300 MHz 20 GB/4 MB | IRIX 64 6.5 | +1.085 | 608 | 35 |
| node III GUP LINZ Austria | INTEL PII 350 MHz 128 MB/512 KB | LINUX 2.2.13 | 1.191 | — | — |
| node IV GUP LINZ Austria | MIPS R12000 400 MHz 64 GB/8 MB | IRIX 64 6.5 | 2.214 | 918 | 23 |
| node V RIST SALZBURG Austria | ALPHA EV6 21264 500 MHz 512 MB/4 MB | OSF I V 5.0 | 3.894 | 592 | 36 |

First of all one must realize that the partitioning scheme only works properly for realistic cases where each of the nodes actually has to contribute to the net result. Therefore a lower threshold value below that a node would become completely redundant need not be considered. Furthermore, the feasibility to the different memory capabilities of the various cluster nodes is also known beforehand. From this it follows that if a certain molecular system once has been successfully prepared for a WAN cluster computation, it is also guaranteed that there will be no memory bottlenecks affecting the parallel run.

The actual time consumption for the parallel region of each cycle will always be determined by the latest incoming host showing up at the top positions in the graph. It is clearly seen that the initial Speed Factors were far from optimal, but iteration 2 and 3 could improve the load distribution significantly — by decreasing Speed Factors of nodes IV and V and increasing the one on node II. Throughout the entire 25 cycles we see a permanent deviation from stable, balanced cluster conditions. This is caused by node II which possesses two different kinds of CPU, either MIPS R10000/200 Mhz or MIPS R12000/300 Mhz, from which one is always taken arbitrarily when PVM re-evokes a process. Nevertheless the dynamic feature of SWLB always manages to immediately change the

load partitioning scheme and bring the cluster nodes back to a more effective load-distribution. On node III a second process becomes active around iteration 13. This is an automatically started backup program that usually runs at 100% CPU occupation. The best indication for how useful selfregulated dynamic SWLB might become in certain situations is that the cluster operates at almost the same net speed as early as in the 15th iteration.

4 Conclusion

Dynamic and auto-regulated runtime-interference to cluster applications — e.g. by employing *Dynamic Speed Weighted Load Balancing* — may help to significantly improve the parallel performance of the application. This is especially usefull when the grid-application performs cyclic iterations and the cluster is of heterogeneous character and many different events may occur as side processes (such as is often the case in WAN clusters). The described mechanism is operational in the context of Quantum Chemistry calculations with GREMLIN but the concept is very general and thus may be applied to similar problems where load partitioning modules are periodically re-evoked.

Acknowledgments

The author would like to thank Prof. Zinterhof from RIST⁺⁺ Salzburg, Prof. Volkert and Dr. Kranzlmüller from GUP Linz Dr. Romaric David from ICPS Strasbourg and Prof. Zerbetto from Ciamician Bologna for providing access to their supercomputer facilities.

References

- [1] Allen, G., Foster, I., Karonis, N., Ripeanu, M., Seidel, E., Toonen, B.: Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. Proc. SC. 2001 (2001) 62
- [2] Höfinger, S., Steinhauser, O., Zinterhof, P.: Performance Analysis and Derived Parallelization Strategy for a SCF Program at the Hartree Fock Level. Lect. Nt. Comp. Sc. **1557** (1999) 163–172 63, 64, 66
- [3] Obara, S., Saika, A.: Efficient recursive computation of molecular integrals over Cartesian Gaussian functions. J. Chem. Phys. **84** (7) (1986) 3963–3974 64
- [4] Höfinger, S.: Balancing for the Electronic Structure Program GREMLIN in a Very Heterogeneous SSH-Connected WAN-Cluster of UNIX-Type Hosts. Lect. Nt. Comp. Sc. **2074** (2001) 801–810 65
- [5] Höfinger, S.: Dynamic Load Equilibration for Cyclic Applications in Distributed Systems. Lect. Nt. Comp. Sc. **2330** (2002) 963–971 63, 65
- [6] Geist, G., Kohl, J., Manchel, R., Papadopoulos, P.: New Features of PVM 3.4 and Beyond. Hermes Publishing, Paris Sept. (1995) 1–10 65, 66
- [7] Hartree, D. R.: Proc. Camb. Phil. Soc., **24** (1928) 89 66
- [8] Fock, V.: Näherungsmethoden zur Lösung des Quantenmechanischen Mehrkörperproblems. Z. Phys. **61** (1930) 126 62 (1930) 795 66

A Contribution to Industrial Grid Computing

Andreas Blaszczyk¹ and Axel Uhl²

¹ ABB Corporate Research, Wallstadter Straße 59,
68526 Ladenburg, Germany
Andreas.Blaszczyk@de.abb.com

² Interactive Objects-Software GmbH, Basler Straße 65,
79100 Freiburg, Germany
Axel.Uhl@io-software.com

Abstract. The current trends related to engineering computations in a distributed industrial environment are presented. The key feature is outsourcing the complex numerical computations and making them available to end users via Intranet. The implementation technology is based on the model-driven development of Java components in an environment including Windows-clients and multiprocessor Linux servers running PVM/MPI. A pilot application from electrotechnical industry is presented.

1 Introduction

The development of computational environments based on Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) significantly influenced the range of numerical applications used by industrial companies. In the mid 1990s the clustering of UNIX-based workstations to speed-up engineering applications was an important milestone in introducing simulation into the design process [1]. It has been regarded as a cost efficient alternative to multiprocessor high-end computers.

After the first successful attempts to create such high performance environments it turned out that their maintenance locally at the production sites is highly inefficient. Particularly, the permanently changing software and hardware components prevent the end users from keeping the complex numerical applications up-to-date.

On the other hand the growing performance of communication networks provides an opportunity of outsourcing simulation components and offering them as a service available via Intranet [2]. This is in fact a concept included in the Grid computing initiative that has recently become a research focus of the academic community [3].

In this paper we describe a pilot application that has been implemented in ABB Corporate Research in the years 2000-2002. It has been aimed at creation of a new service available to ABB designers via Intranet. This service includes the computations of 3D electromagnetic fields for dielectric design of high voltage arrangements. This application is based on Pro/Engineer (used also as mesh generator and post-processor) as well as the ABB-in-house boundary element solver Polopt [4].

2 Architecture and Requirements

Fig 1 shows a distributed simulation environment of a large (or medium-size) industrial corporation. Several business units (BUs) are connected to one or few central sites that provide simulation services. The computers in the business units are graphical workstations running CAD-systems and the analysis modeling tools. The numerical simulation jobs are created on these workstations and the appropriate data are exchanged with a solver running on the central server computer via the network. The server computers are multiprocessor Linux clusters with PVM/MPI software.

The central sites hosting the solvers and other simulation services are usually located in sites that are central to the organization e.g. research labs or computing centers (CRCs). The business units can outsource the simulation resources to the central sites - saving tens of thousands of Dollars yearly and maintaining access to the latest simulation technologies - without creating the required infrastructure locally. The central sites maintain this kind of expertise anyway as it is needed for advanced research projects. The central sites also provide a link to the Internet: in particular the access to the future Grid Computing services.

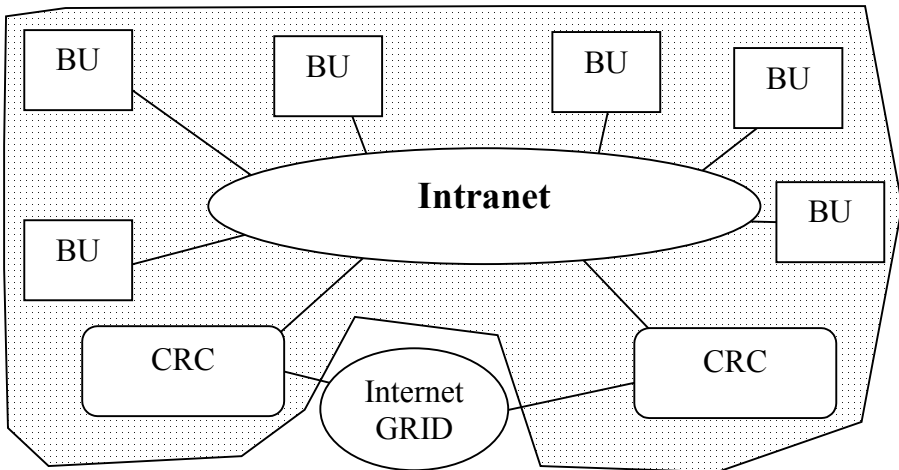


Fig.1. Architecture of an industrial Grid

The requirements of the end users are as follows:

- The remote solver is activated directly from the CAD-session; the whole process is transparent for the user: The data transfer, launching the remote solver, and activation of MPI/PVM happens automatically. Optionally, the user can specify the number of processors.
- The whole solution procedure is accomplished within few minutes (no longer than 1 hour). Queuing the job on the server (for several hours) is not acceptable.

- The data generated by the solver remain on the server except for the final result file that has to be loaded into the CAD-system on the client side. Evaluation and visualization tools running on the client can access the data on the server.
- Authentication of users is required, but should be performed only once during a session that includes many solver jobs or data evaluations.

3 Basic Technologies

3.1 Java

Java is well established as a computing platform for Internet related applications. The following features make Java particularly suitable for implementing software components used in distributed simulation environments:

- It contains internal mechanism for client-server interaction based on the Remote Method Invocation (RMI) package.
- Numerical computations can be done with a good performance [5]. This is essential for engineering applications where pre- and post-processing of numerical data is required. Some less time consuming, non-parallel numerical computations can be performed by client computers, which may significantly reduce the communication effort.
- Java can be used for the creation of all engineering GUI-components including 3D-graphics.
- Portability between Linux and Windows is ensured.
- The automatic upgrade of the client software installation can be implemented easily.

3.2 Linux Cluster Computing

Linux clusters have been identified as the most cost-effective platform for high performance computing since the end of the 90s (www.beowulf.org). The strong growth of cluster computing is reflected in the latest edition of the top-500 list of the world's most powerful computers, see: www.top500.org.

An important component deciding about the performance of parallel applications is the internal network connecting the cluster nodes and the appropriate PVM/MPI implementation. The network applied in most industrial installations is fast Ethernet. Installing Scalable Coherent Interface (SCI) or Myrinet solutions can significantly improve the network performance. However, this may be associated with considerable additional costs (up to 50% or more). The selection of the configuration for industrial clusters is strongly dependent on the applications. For example, in case of PVM/MPI-based boundary element formulations it is essential to have enough memory for the in-core storage of the dense matrix part calculated on each node [1], [4]. Therefore, for this type of applications we tend to invest more into the computer memory than into speeding up the interconnect network.

3.3 Model Driven Development of Software Components

A software component for the client-server interaction between a CAD user and the computer engine strongly depends on simulation procedures applied by designers. The simulation tools, items for evaluation, as well as designer requirements, change from business to business and consequently there is a need to adapt the software components to the changing environment. This problem can be effectively managed by describing the software components in *models* and use the *Model-Driven Architecture* (MDA) as defined by the Object Management Group (OMG) to transform these models into executable implementations with little effort and in portable ways. This approach is described in detail in [6, 7].

The general idea of the MDA is based on automating transformations between models using tools. As a special case, MDA considers source code as a model, because it complies with formal syntactical and semantical definitions. A typical model transformation in the scope of the MDA thus is to generate source code from, e.g., a UML model.

Models are instances of metamodels. Furthermore, the system specifications expressed in a model may exhibit dependencies on properties provided by *platforms*. The less such dependencies a model has, the more platforms can be found that supply what the model depends on, and the less dependent the model is on a particular one of these platforms. For example, the Java source code depends on the Java programming language and the interfaces provided by the Java Development Kit (JDK). However, it is largely independent of the operation system platform, because there are several on which Java programs can be executed without changes.

Models can be *annotated* for specific transformations. The annotations thus may have to have knowledge about the target platform. On the other hand, annotations can therefore be used to keep the model clear of dependencies to those specificities of the target platform, resulting in increased model portability.

A tool that implements the MDA concepts is ArcStyler (www.io-software.com). It leverages the UML modeling tool Rational Rose (www.rational.com) for the creation, visualization and editing of UML models, and provides the infrastructure for specifying and executing the required model transformations. With ArcStyler, the transformations can easily be customized and extended as needed, thus realizing the promise of providing the models with portability and longevity.

ArcStyler thus provides a future-oriented platform for the model driven development of components of simulation environment; in particular, they allow the following goals to be achieved:

- The developed model can be very quickly extended and reused for other applications. A crucial feature is that the source code (written, e.g., in Java) can be automatically regenerated for a changed model. The parts of code written manually are protected and will not be lost during regeneration.
- A UML-model is not dependent on the current technology. Based on such a model one can automatically generate the source code for any given implementation technology. For example, the remote access to simulation resources developed for the pilot application (shown in the next section) is currently based on Java/RMI, but by applying other model transformations/cartridges in MDA tools the implementation

can be automatically changed to another technology like CORBA. In this way we are independent of rapidly changing Internet technologies and can focus on the functionality of our simulation environment.

3.4 Grid Computing

The Grid computing concept includes a collection of technologies that enable sharing hardware and software resources in a networked environment. The reference software for building Grid infrastructure and applications is the Globus Toolkit (www.globus.org). It includes implementation of services like security, communications, data and resource management [3].

The application presented in the next section is not yet based on the Grid-software. We intend to include the Globus Toolkit in the second stage of the project. We expect that the MDA-approach explained above will enable a smooth transition to the new implementation technology.

4 Pilot Application

According to the presented architecture, the requirements and the available basic technologies the client-server model shown in Fig. 2 has been developed. The following programs are included in the first implementation:

- submit-job client
- evaluation&visualisation client
- client proxy
- server.

Each of these programs is executed by launching a Java Virtual Machine (JVM). The first three ones are running on a Windows client the last on a Linux cluster.

The submit-job client program is activated directly from a CAD-session. It sends the input file generated by the CAD-system to the server, launches the remote solver together with PVM/MPI, and receives the result file. The first time a job is submitted a user session is opened. The context of the session including login name, password, project name, and the selected application is obtained from the user. This ensures a restricted access to the user data on the server and enables the selection of the appropriate solver options. For all subsequent submit-job runs the context is retrieved from the client proxy.

Typically, the user submits several jobs that are followed by post-processing sessions in the CAD-system. The evaluation&visualization client is required only for services that are not available in the CAD-system. The evaluation of the business specific design criteria is one of such services. Another one is visualization of specific simulation results using the 3D VRML-technology. Both services are based on data created by the previous jobs and stored on the server. The evaluation&visualization client provides these services together with an overview of the directories and files stored on the server. Additionally, the access to documentation has been integrated into this client.

The client proxy is a program running in the background. Its main role is to keep references to the remote server objects while the other two client programs are not active. Every time the submit-job or evaluation&visualization client is started, a connection to the proxy is established and the remote server objects can be referenced. In this way, an active server session with its context can be resumed. Additionally, the client proxy is responsible for monitoring any running jobs.

The server program is responsible for the creation and administration of remote objects like server, user, session, application, project, and job, see Fig. 3. The corresponding classes specify basic services for authentication, communications, and data administration. Multiple clients can transparently access these services at the same time. An important server component is the solver wrapper (the Solver-item in Fig. 3). It is responsible for launching and stopping the solver, monitoring progress as well as for setting up the appropriate MPI/PVM configuration.

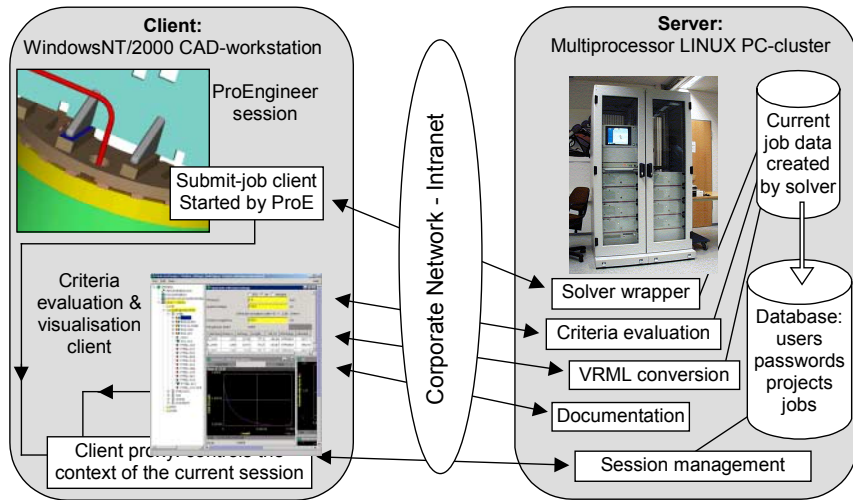


Fig. 2. Client-server model for accessing remote resources from a CAD-environment

The client-server model has been designed in UML using Rational Rose and Arc-Styler. A part of its graphical representation is shown in Fig. 3. The generated source code is based on pure Java: the user data are stored as serialized objects; Remote Method Invocation (RMI) is used for communications; solvers together with PVM/MPI environments are launched from Java using the Runtime-utility, the monitoring is based on files created by the solver.

Using this client-server model, the dielectric design application based on the ABB solver Polopt has been implemented. However, the same model is applicable to any other application and solver. We intend to use this model for dielectric optimization and electromechanical design in the near future. For the new applications additional customized client modules as well as a new solver wrapper on the server side must be designed. In general, the applications are independent from the technology used for implementation of the client-server model. They can be dynamically integrated with this model.

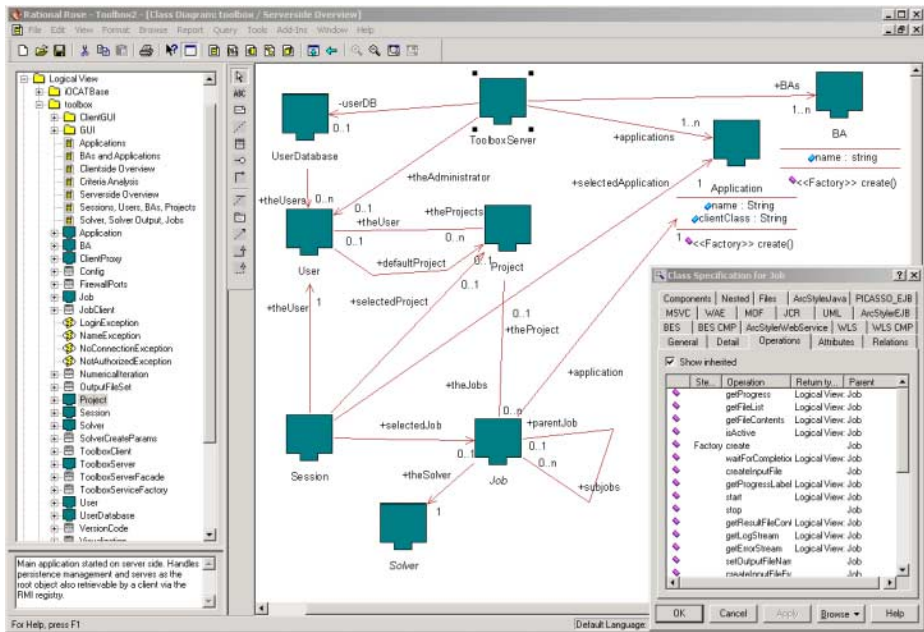


Fig. 3. Graphical representation of the client-server model created by ArcStyler and Rational Rose. This example shows the server-side overview. The dark squares represent remote objects created by the server and the lines represent the relations between them. A double-click on each of these squares and lines opens the corresponding ArcStyler specification wizard as shown in the right bottom corner

5 Conclusions

There is a strong trend in industrial computing to outsource the high performance environments out of the end user locations. The specialized application providers should maintain the advanced PVM/MPI based systems for parallel computations together with the appropriate solvers and offer them as a service in an intranet. This has been implemented in the pilot project described above which can be regarded as the first step to Grid computing in an industrial environment.

We expect a surge of interest for this type of services offered via the network within the next few years. In order to satisfy the requirements of the end users, a considerable effort should be invested in creating customizable distributed simulation environments that will connect the CAD-world of engineers with the parallel, high performance computers running the advanced numerical applications. Using a model-driven (MDA) approach for the construction of these distributed software systems improves portability and thus longevity of the system specifications that are developed. We want also to emphasize that the MDA-approach enables a smooth transition to the Grid-technologies since the developed UML models can be reused.

These trends will also influence the PVM/MPI environments. Their integration into Grid- and Java-platforms could be crucial for the efficiency of parallel applications launched and monitored from the remote clients.

References

- [1] Blaszczyk, C. Trinitis: Experience with PVM in an Industrial Environment, Lecture Notes on Computer Science Vol. 1156, Parallel Virtual Machine - EuroPVM'96, Munich, Proceedings pp. 174-179.
- [2] Blaszczyk, H. Karandikar, G. Palli: Net Value: Low Cost, High Performance Computing via Intranet. ABB Review, 2002/1, pp. 35-42.
- [3] Foster, C. Kesselman, S. Tuecke: The anatomy of the Grid. Enabling scalable virtual organizations. International J. Supercomputer Applications, 15(3), 2001.
- [4] N. de Kock, M. Mendik, Z. Andjelic, A. Blaszczyk: Application of 3D boundary element method in the design of EHV GIS components, *IEEE Magazine on Electrical Insulation*, May/June 1998 Vol.14, No. 3, pp. 17-22.
- [5] J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, M. Snit, R.D. Lawrence: Java programming for high performance numerical computing. IBM Systems Journal, Vol 39, No 1, 2000.
- [6] Thomas Koch, Axel Uhl, and Dirk Weise. Model-Driven Architecture, January 2002. URL: <http://cgi.omg.org/cgi-bin/doc?ormsc/02-01-04.pdf>.
- [7] The Object Management Group (OMG). Model Driven Architecture: The Architecture of Choice for a Changing World, 2001. URL <http://cgi.omg.org/cgi-bin/doc?ormsc/01-07-01>

Parallel Computing for the Simulation of 3D Free Surface Flows in Environmental Applications

Paola Causin and Edie Miglio

MOX – Modeling and Scientific Computing, Dipartimento di Matematica
“F. Brioschi” Politecnico di Milano, via Bonardi 9, 20133 Milano, Italy
{Paola.Causin,Edie.Miglio}@mate.polimi.it,
<http://www.mox.polimi.it>

Abstract. The numerical simulation of 3D free surface flows in environmental fluid-dynamics requires a huge computational effort. In this work we address the numerical aspects and the computer implementation of the parallel finite element code *Stratos* for the solution of medium and large scale hydrodynamics problems. The code adopts the MPI protocol to manage inter-processor communication. Particular attention is paid to present some original details of the software implementation as well as to show the numerical results obtained on a Cray T3E machine.

1 Introduction and Motivation

Free surface flows are encountered in various natural phenomena, such as tidal flows, large water basins and river courses. The huge computational effort required by a full 3D numerical simulation has led in the past to developing simplified 1D and 2D models. These latter models are nowadays well established, both in terms of a sound mathematical formulation and of a robust numerical implementation (see [1], [2] and [6]). Nonetheless, many situations of relevant interest occur that require a more accurate physical description of the problem in order to model all the significant phenomena.

The so-called *Quasi-3D Shallow Water Equation* model (Q3D-SWE) considered in the present work enriches the 2D models by a proper selection of 3D features (*e.g.*, a velocity field that depends on all the three spatial variables), with a computational cost not severely exceeding the cost of pure 2D models.

However, it is a fact that even adopting the Q3D-SWE model, hydrodynamics problems in environmental applications inherently require a very large number of unknowns. This fact, considering furthermore the fairly long characteristic times involved in these problems (often several days), lead to exceedingly expensive computations. Consider as an example the simulation of the tides in the Venice Lagoon whose numerical results will be presented in Sect.4. Such simulation, spanning the tidal cycle of two days, has been carried out using 20 layers in the vertical direction and about 31500 triangles in the horizontal plane covering an area of 1800 km², for a total of approximately of 2 millions degrees of freedom

with a time step of 15 minutes. These latter figures clearly demonstrate that a parallel approach is in order. The results presented in this work have been obtained running the code *Stratos* on a T3E Cray machine using up to 8 processors. The code adopts the MPI Message Passing Protocol in order to guarantee the maximum portability; it is indeed planned to run the simulations in the future on a cluster of PCs.

The paper is organized as follows: in Sect.2 we present the Q3D-SWE mathematical model; in Sect.3 we address the finite element discretization of the Q3D-SWE model and the parallel computer implementation; in Sect.4 we present the numerical results, while in Sect.5 we draw the conclusions and we present the future directions of the work.

2 The Q3D-SWE Mathematical Model

The Q3D-SWE model is derived from the three dimensional incompressible Navier-Stokes equations by integrating the continuity equation along the vertical direction,

Before presenting the model, we need to set up some basic notations. Indicating by $\hat{\Omega}$ a parallelepipedon which includes the water body for the whole time interval of interest and denoting by Ω the base of this parallelepipedon, the bottom surface and the free surface of the domain are described by the bottom bathymetry $z = -h(x, y)$ and by the elevation of the free surface $z = \eta(x, y, t)$, respectively. The total depth of the fluid at a generic point (x, y) is denoted by $H(x, y, t) = h(x, y) + \eta(x, y, t)$. Under the assumption of *long wave phenomena*, vertical accelerations can be neglected, leading to the following *hydrostatic approximation* for the pressure

$$p = p_0 + \rho g(\eta - z), \quad (1)$$

where p_0 is the given atmospheric pressure and g is the gravity acceleration. The Q3D-SWE model reads [4]:

$\forall t \in (0, T]$, find (\mathbf{u}, w, η) such that

$$\begin{cases} \frac{D\mathbf{u}}{Dt} = -g\nabla_{xy}\eta + \nu\frac{\partial^2\mathbf{u}}{\partial z^2} + \mathbf{f}, \\ \frac{\partial\eta}{\partial t} + \nabla_{xy} \cdot \left(\int_{-h}^{\eta} \mathbf{u} dz \right) = 0, \\ \frac{\partial w}{\partial z} + \nabla_{xy} \cdot \mathbf{u} = 0, \end{cases} \quad (2)$$

where ∇_{xy} denotes the spatial gradient in the horizontal plane $D/Dt = \partial/\partial t(\cdot) + \mathbf{u} \cdot \nabla(\cdot)$ is the Lagrangian time derivative, ν is the vertical eddy viscosity coefficient, $\mathbf{u} = (u, v)^T$ is the horizontal velocity, w is the vertical velocity, \mathbf{f} is the external force vector (typically the Coriolis force). Notice that relation (1) has been employed to express the gradient of the pressure in the momentum equation as a function of the sole elevation gradient. System (2) is completed

by a proper set of initial and boundary conditions that specify the elevation or the velocity of the fluid on the boundary of the domain (water-water boundary or water-solid wall boundary, respectively). Wind effects as well friction effects can be also accounted for by forcing to a given value the tangential component of the normal stress on the free surface or on the bottom, respectively.

3 Parallel Implementation of the Q3D-SWE Model

In this section we address the issue of the discretization of model (2) and we focus on the aspects concerning its implementation on a computer with parallel architecture.

3.1 Discretization and Partitioning of the Domain

The discretization method adopted in this work decouples the vertical and the horizontal directions. This procedure is a natural consequence both of the mathematical form of the Q3D-SWE equations and of the geometrical characteristics of the computational domain. Namely, the parallelepipedon $\hat{\Omega}$ is sliced in the vertical direction into several horizontal parallel layers, with possible different heights. An unstructured triangulation is used in the horizontal plane xy , in order to provide an accurate description of the irregular planar geometries that frequently occur in these problems. The replication of the same triangular grid in the middle point of each layer creates a mesh of right-angle prisms. At each time level, a prism may lay entirely or partially under the free surface level or may be dry. Tracking the free surface position is thus reconducted to managing efficiently the emptying and filling of the prisms. Very irregular bathymetries of the bottom, as obtained from experimental measurements, can be handled as well. In Fig. 1 we show the very irregular bathymetry of the bottom (left) and the finite element triangulation in the xy plane (right) for the Venice Lagoon. Notice how the maximum density of the triangulation is set in correspondence of the lagunar isles and of the channels that exchange the fresh water from the Adriatic Sea to the Lagoon, where more resolution is needed.

The basic prismatic cell one layer deep in the z direction may be regarded as the most *fine grained* decomposition level of the problem. The granularity of the partition is increased by agglomerating all the layers of an element into a column. Each column of cells is then mapped to a processor. This choice significantly reduces communication costs by increasing locality. The package Metis [5] has been used to partition the mesh in the horizontal plane, since each column is unambiguously identified through a tag that is the number of its base triangle in the xy plane. Fig. 2 shows an exploded 3D view of the sub-domains assigned to each of the 8 processors used in the simulation.

3.2 Finite Element Approximation of the 3D-SWE Model

We refer to [3] and [4] for a detailed discussion of the finite element spaces adopted for the approximation of the Q3D-SWE model. Here we limit ourselves

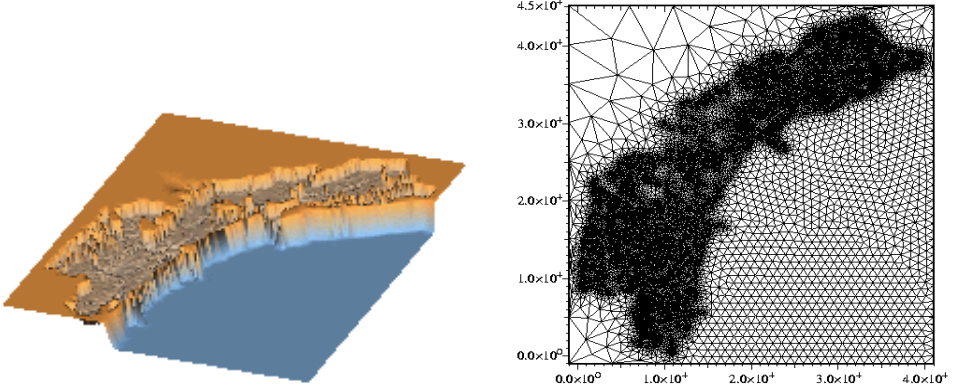


Fig. 1. The complex bathymetry of the Venice Lagoon (left) and the mesh in the horizontal plane (right)

to present the aspects that are relevant to the parallel computer implementation. Among the main issues to deal with, we mention:

1. the management of unstructured grids in a distributed architecture that demands for additional connectivity structures;
2. the existence of several procedures that require a search procedure in all the elements of the domain (as happens in the computation of the starting point of the characteristic line, that will be discussed in the following) must be suitably specialized to the multiprocessor environment;
3. the objective of obtaining the maximum efficiency of the code must be pursued avoiding undeterministic behaviors that may often occur during the interprocessor message passing phase in a non-correctly planned algorithm.

On this basis, it has appeared convenient to adopt an object-oriented programming concept, so to exploit the intrinsic links of data and underlying structures in association with the tasks (methods) they have to carry out in the parallel environment. The introduction of abstract data types can be applied to describe geometric objects such as elements and edges as well as to collect a set of tasks such as creating the element matrix or updating the variables. Let us consider two important classes of mesh objects: the *element class* and the *edge class*. Each element object contains the required connectivity data (its name, the names of its edges and vertices and the names of its adjacent elements), the *ids* of the processors that own the element itself and the adjacent elements, the geometrical data (area, coordinates of the circumcenters and of the vertices) and the numerical values of the fields whose degrees of freedom are associated with the element, namely, the elevation of the fluid and the vertical velocity w . Let us consider as an example Fig.3, where the area of a larger domain is depicted and let us focus the attention on the elements labeled by ⑤ and ⑭. The abstract element class and the corresponding instances for elements ⑤ and ⑭ are represented in Fig.4 (left). The second class of objects, edge objects, retain the

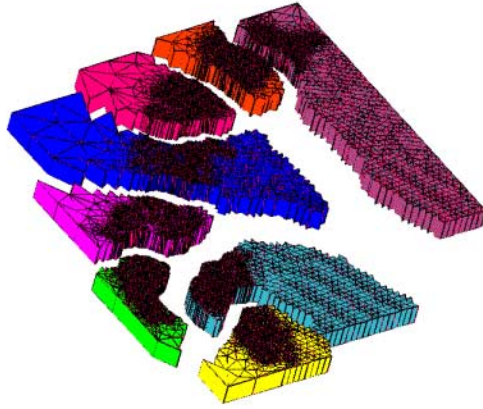


Fig. 2. An exploded view of the domain partitioning for the simulation of the Venice lagoon

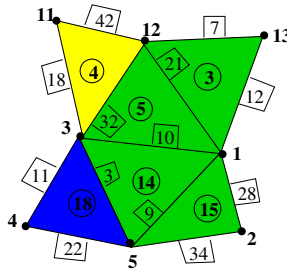


Fig. 3. Each color in the figure identifies a different processor. Circle: element numbers; square: edge numbers; solid points: vertex numbers

information regarding the number and the thickness of the active layers that at each temporal step describe the actual shape of the surface of the fluid and the bathymetry. Moreover, they represent the fundamental geometrical structure for the horizontal velocity in the xy plane. In fact the Raviart-Thomas finite elements of lowest degree yield a finite volume-like approach that uses as degrees of freedom the fluxes of the unknown quantity across the edges of the triangulation. Edge objects are assigned the fundamental role to manage inter-processor communication; the element class does not perform message passing but it delegates communication to the edge class. The edges collect and broadcast data to be sent, distribute received data and maintain all the information required for a proper organization of the message passing itself. To improve efficiency, while the interface edges perform *non-blocking* message passing actions, internal edges overlap computation on processor-owned data. The abstract edge class and the corresponding instances for edges [32] and [3] are represented in Fig.4 (right).

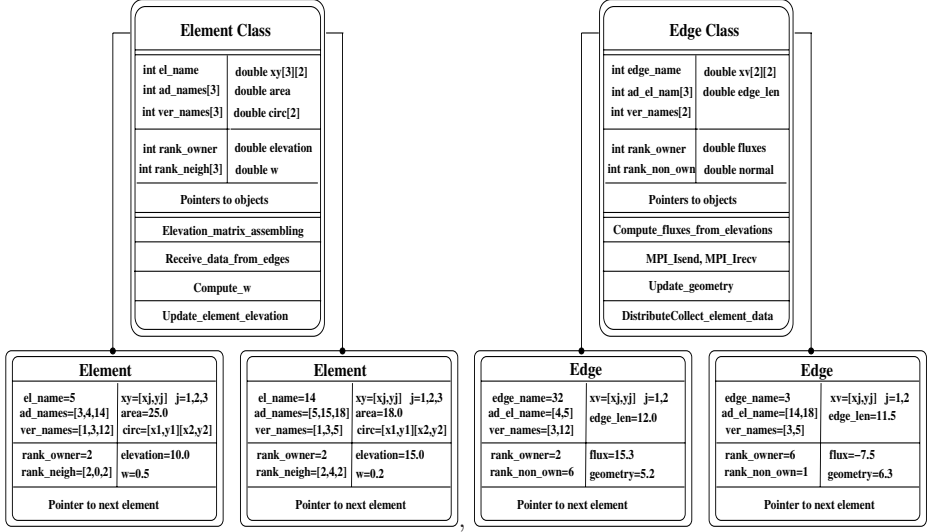


Fig. 4. The element class with its attributes and methods and two instances (left) and the element class with its attributes and methods and two instances (right)

Since the data stored in the edge objects are very often accessed by the algorithm, a number of short-cuts have been adopted to reduce access times. One example is represented by the additional set of pointers provided in order to obtain a quick access to specific entries in the edge list. The pointers allow to restrict sorting operations to specific subset of possible candidates.

3.3 Computation of the Lagrangian Time Derivative

In order to ensure numerical stability when dealing with the convective terms in (2), a Lagrangian approach is used. This turns out to add a number of complications in the parallel implementation. Denoting by \mathbf{x}_{mi} the midpoint of a vertical face of a prism, the Lagrangian discretization of the convective term reads:

$$\frac{D\mathbf{u}}{Dt}(t_{n+1}, \mathbf{x}_{mi}) \simeq \frac{\mathbf{u}(t_{n+1}, \mathbf{x}_{mi}) - \mathbf{u}(t_n, \mathbf{X}(t_n; t_{n+1}, \mathbf{x}_{mi}))}{\Delta t}.$$

where $\mathbf{X}(s; t, \mathbf{x})$ is the solution of the following problem:

$$\begin{cases} \frac{d\mathbf{X}(s; t, \mathbf{x})}{ds} = \mathbf{V}(s, \mathbf{X}(s; t, \mathbf{x})) \text{ for } s \in (0, t), \\ \mathbf{X}(t; t, \mathbf{x}) = \mathbf{x}. \end{cases} \quad (4)$$

From a geometric point of view $\mathbf{X}(\cdot) = \mathbf{X}(\cdot; t, \mathbf{x})$ is the parametric representation of the streamlines: *i.e.* $\mathbf{X}(s; t, \mathbf{x})$ is the position at time s of a particle which has been driven by the field $\mathbf{V} = (\mathbf{u}, w)$ and that occupied the position \mathbf{x} at time t .

Equation (4) can be solved using an explicit Euler method (see Fig. 5) which gives the following approximation of $\mathbf{X}(t_n; t_{n+1}, \mathbf{x})$:

$$\mathbf{X}(s; t, \mathbf{x}) = \mathbf{x}_{mi} - \mathbf{V} \Delta t \quad (5)$$

where the velocity \mathbf{V} is the velocity evaluated at \mathbf{x}_{mi} at time t_n . From the left part of Fig. 5 it is evident that the Euler method may produce very bad results in presence of strongly curved streamlines unless a very small time step is used. To overcome this problem we solve problem (4) using a *composite* Euler scheme: the time step Δt is split into N_δ substeps (generally $N_\delta \simeq 4$ and in each substep a simple Euler scheme is used (see right part of Fig. 5). In this way the streamlines are better approximated and if the number of substeps is properly chosen for the problem at hand the computed streamlines do not cross solid boundaries.

It is clear that the characteristic method relies strongly upon an efficient searching algorithm, since we have to find the element in which the foot of the characteristic line (denoted by \mathbf{X}) starting from the generic point \mathbf{x}_{mi} is located. Let us briefly describe the searching algorithm. The searching algorithm can be conveniently decoupled into a search along the vertical direction, which is discretized using a structured grid and does not present difficulties and a search in the xy plane that requires more case. In particular, we proceed as follows: let $\tilde{\mathbf{x}}_{mi}$ and $\tilde{\mathbf{X}}$ be the projections of the starting point, \mathbf{x}_{mi} , and of the foot of the characteristic, \mathbf{X} , on the xy plane. The point $\tilde{\mathbf{x}}_{mi}$ is the middle point of the edge ℓ_i of the 2D mesh. First of all we check if $\tilde{\mathbf{X}}$ belongs to one of the two triangles sharing the edge ℓ_i , in which case the procedure ends. Otherwise we take one of these two triangles as the starting element (we will denote it with T) and we go through the following steps

1. build the segment s joining $\tilde{\mathbf{x}}_{mi}$ and $\tilde{\mathbf{X}}$;
2. determine the edge ℓ_t of T crossed by s and find the adjoining element (T') of T sharing with T the edge ℓ_t .
3. if the element T' is owned by a different processor, set $\tilde{\mathbf{X}} = \tilde{\mathbf{x}}_{mi}$. Otherwise check if $\tilde{\mathbf{X}}$ belongs to T' , if not, iterate the procedure.

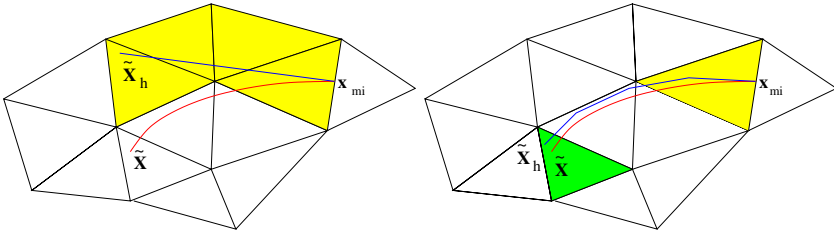


Fig. 5. Representation of the discretization of the streamlines using the Euler method: simple Euler method (left), composite Euler method (right)

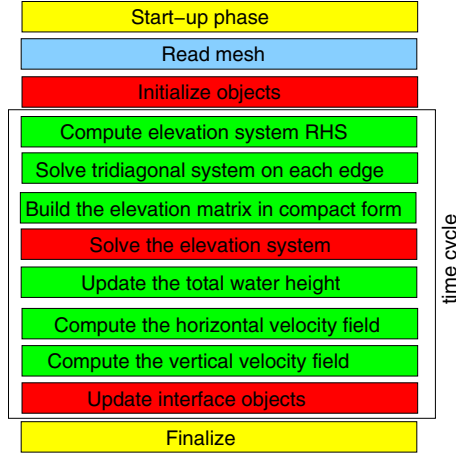


Fig. 6. Block diagram. Colors indicate the following: yellow: initialization phase; red: interprocessor communication; green: parallel work; cyan: I/O phase

3.4 The Algorithm Implemented in the Code *Stratos*

In Fig. 6 it is represented a block diagram that illustrates the phases of the computation, showing when interprocessor communication is needed. In the **start-up phase**, the processor conventionally labeled with `proc-id=0` (master processor) reads the common data (the values of the physical parameters, the position of the layers in the vertical direction, the values of the parameters required by the computation, such as the time step) and sends them to the other processors initiating a global communication. Next, the master processor reads the mesh and the corresponding connectivity structures and prepare packs of data to be sent to the other processors. A non-blocking communication is then initiated and the collected data are appropriately parsed to build local element and edge objects. The following three steps (green boxes) are intensive local computation blocks aimed at building the coefficient matrix and the right hand side of the linear system on the elevation unknowns (see [4] for the mathematical details). The solution of such system, that is performed using modern iterative methods based on Krylov-spaces iterations, relies on the Aztec library [7]. Special connectivity structures that are needed for the Aztec library to work have been specialized for the problem at hand. In turn, the programming strategies of the Aztec library have been thoroughly studied and have been considered as a guideline to improve the efficiency of our work. After solving the linear system, the velocity field can be recovered. Along the vertical direction piece-wise continuous polynomials are adopted. This implies a coupling between the points of the same column, but since the entire column is owned by the same processor, this process is completely *local*. The velocity components in the xy plane require instead interprocessor communication between the edges located along inter-

processor boundaries. These edges are duplicated, the flux is computed twice by two different processors and then averaged (this is due to a possible different computation of the Lagrangian derivative, see below). In order to set up unambiguous procedures for interprocessor communications, a criterion to determine the formal ownership of the edges has to be established. An interface edge is considered owned by the processor that has been assigned the base triangle with the smallest global numbering between the two elements that share that edge. The concept of ownership of an edge turns out to be also useful to determine the orientation of the unit normal vector to the edge; indeed, after establishing, for example, that the edge unit normal vector is oriented from the non-owner element to the owner element - this convention being necessary to obtain a coherent reconstruction of the velocity field from the fluxes -, this information may be retrieved independently by each processor from its own connectivity database built at the beginning of the computation. Eventually, at the end of each time step, each processor saves its local results for the successive graphical post-processing. A separated parallel code handles the visualization of the data.

4 Numerical Results

To assess the properties of scalability of the code, we have performed the 3D Gaussian hill test case using up to 8 processors (the maximum number we were allowed) on a Cray T3E machine. The results indicate a speed-up factor of the code that is almost linear. The following figure show instead the computation on the Venice Lagoon. In Fig.8 (right) it is shown the water elevation, while in Fig.8 (left) it is shown the the velocity in the Venice Lagoon.

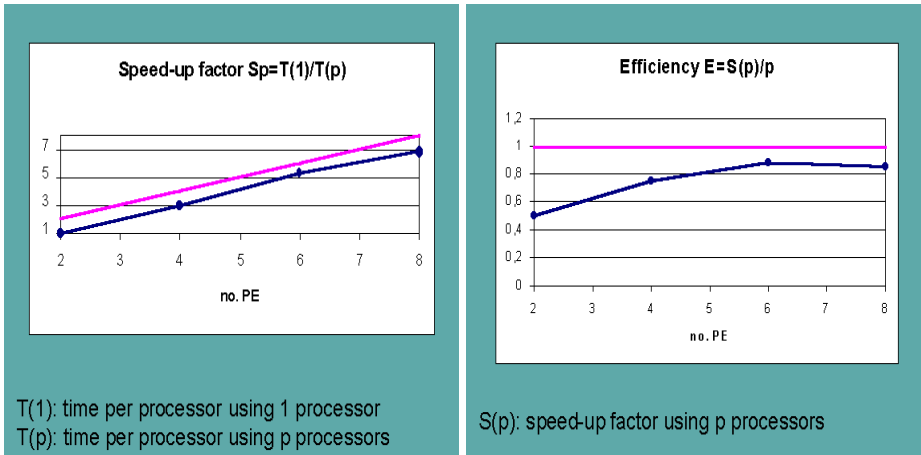


Fig. 7. Speed-up factor and efficiency for the 3D Gaussian hill test case on Cray T3E

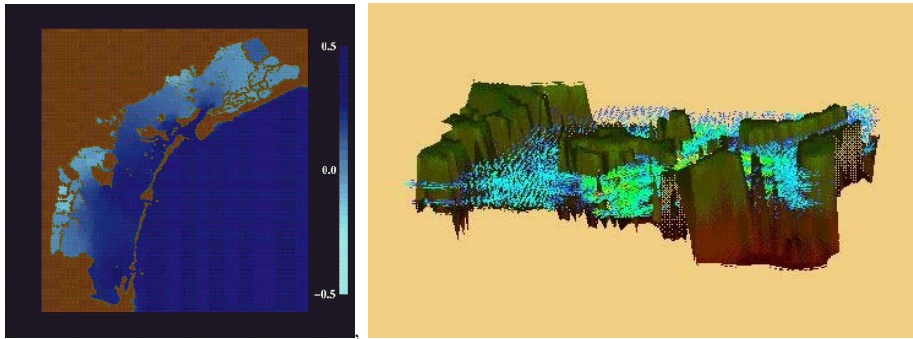


Fig. 8. Three dimensional velocity field in a region of the Venice Lagoon (right)

5 Conclusions

In this work we have addressed the numerical aspects and the MPI computer implementation of the Q3D-SWE model for the simulation of free surface flows in environmental problems. The presence of the free surface requires to deal with a number of issues that are not commonly present in fixed domain fluid-dynamics problems. In this sense, some original details of the implementation have been discussed. Future research will be devoted to develop and implement models able to couple free surface flows with permeable soils through which the surface water can filter. This aspect is of relevant importance, especially in presence of pollutants.

References

- [1] Agoshkov, V. I., Ambrosi, D., Pennati, V., Quarteroni, A., Saleri, F.: Mathematical and Numerical Modeling of Shallow Water Flow, *Computational Mechanics*, 11 (1993). 78
- [2] Barragy, E. J., Walters, R. A.: Parallel iterative solution for h and p approximations of the shallow water equations, *Advances in Water Resources*, 21 (1998). 78
- [3] Causin, P., Miglio, E., Saleri, F.: Algebraic factorizations for non-hydrostatic 3D free surface flows, to appear on *Computing and Visualization in Science* 5 (2) (2002), pre-print available at the web-site: <http://mox.polimi.it>. 80
- [4] Fontana, L., Miglio, E., Quarteroni, A., Saleri, F.: A Finite Element Method for 3D Hydrostatic Water Flows, *Computing and Visualization in Science*, 2/2-3 (1999). 79, 80, 85
- [5] Karypis, G., Kumar, V., 'Metis, A Software Package for Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices', University of Minnesota (1996). 80
- [6] Paglieri, L., Ambrosi, D., Formaggia, L., Quarteroni, A., Scheinine, A. L.: Parallel computation for shallow water flow: a domain decomposition approach, *Parallel Computing*, 23 (1997). 78

- [7] Tuminaro, R.S., Heroux, M., Hutchinson, S.A., Shadid, J.N.: Official Aztec User's Guide: Version 2.1 (1999) [85](#)

Testbed for Adaptive Numerical Simulations in Heterogeneous Environments

Tiberiu Rotaru and Hans-Heinrich Nägeli

Institute of Computer Science, University of de Neuchâtel
Rue Emile Argand 11, CH-2007 Neuchâtel, Switzerland
{tiberiu.rotaru,hans.naegeli}@unine.ch

Abstract. Simulation can reduce the development time of complex applications in science and engineering. We designed HeRMeS, a tool offering dynamic load balancing support which allows to perform numerical simulations on a heterogeneous collection of distributed memory machines. This can serve as a testbed for testing new dynamic data redistribution methods in heterogeneous environments as well as a platform for testing adaptive numerical solvers with built-in heterogeneous dynamic load balancing support.

1 Introduction

Many challenging applications in science or engineering, in domains like fluid dynamics, climate modeling, astrophysical simulations, need to be solved in parallel in order to achieve higher performance. The existence of a number of implementations of the MPI standard such as MPICH or LAM/MPI facilitates now the use of commodity processors for running parallel applications. However, their execution on dynamic and heterogeneous networked computing environments imposes a number of supplementary problems to software designers. Programming in such environments constitutes a much more difficult task as the developers must take into account both the system runtime changes and the application dynamic behaviour. An important characteristic of many complex applications is the algorithmic adaptativity. The mesh based applications implementing parallel adaptive numerical solvers constitute a good example. The adaptive solvers seek to improve the accuracy of the solution by dynamically refining the underlying mesh in regions where the solution error is not satisfactory. Typically, a subdomain of a mesh is associated with a processor and each computation in the local subdomains is performed in parallel for a time-step. The computation is iteratively performed using for each subdomain values computed in the previous step. In the course of a computation the mesh may undergo significant changes as a result of applying refinement or de-refinement operations. The general scheme of an application based on parallel unstructured mesh computations is as described by algorithm 1 (see [1]).

In heterogeneous environments, besides the dynamic changes caused by the adaptive nature of a numerical solver, this type of applications must also react

Algorithm 1 General scheme of a parallel numerical adaptive application.

```

Construct a mesh  $M_0$  that conforms to the input geometry
if not already partitioned then
    Partition  $M_0$  across the  $p$  processors
end if
 $i = 0$ 
repeat
    Assemble a sparse matrix  $A_i$  from  $M_i$ 
    Solve the linear system  $A_i x_i = b_i$ 
    Estimate the error on  $M_i$ 
    if maximum error estimate on  $M_i$  is too large then
        Based on the error estimates, refine  $M_i$  to get  $M_{i+1}$ 
        if partitioning is not satisfactory for  $M_{i+1}$  then
            Repartition  $M_{i+1}$  across the  $p$  processors
        end if
    end if
     $i = i + 1$ 
until maximum error estimate on  $M_i$  is satisfactory

```

to the system modifications, which typically are generated by the competition between processes/applications for system resources.

2 Towards a Generic Simulation Platform

The number of scientific and industrial parallel applications has significantly grown in the last decade. However, the existing software is still characterized by a lack of generalization and standardization, and maybe most importantly, by a miss of adequate documentation. This reduces the possibility of reusing already available codes, many researchers/programmers preferring to write their own programs. On the other hand, developing parallel software for scientific and computational engineering problems requires significant expertise and effort from the part of developers, as the applications tend to involve more and more advanced interdisciplinary knowledge. A key problem that any parallel application must cope with is that of ensuring a fair utilisation of the available system resources at runtime. Developing specialized software modules, particularly a dynamic load balancing module, requires validation for a broad set of test-cases. In many cases the validation is done with respect to some specific applications and/or hardware architectures. The absence of clearly defined reference testbeds for testing diverse dynamic load balancing methods constitutes a limiting factor for the developers. On the other hand, developing a reusable simulation framework may contribute to a substantial reduction of the development costs and permits to address the performance and adaptability requirements of large, complex simulations. Thus, the development time of applications attempting to solve large problems in science and engineering and involving a large number of software, hardware and human resources can be significantly reduced.

Our goal is to provide a general framework incorporating dynamic load balancing support for a large class of applications in the area of scientific computing, respecting their general characteristics, and offering in the same time to the user the possibility to program on top his own application with reasonable effort, without being directly concerned with the dynamic load balancing implementation aspects.

In heterogeneous environments, in addition to the application dependent parameters, the processor related parameters as well as the network related parameters must be taken into account for determining an optimal load distribution, for improving the execution time of the entire application. Specifically, physical parameters of both the interconnection network and the processors, such as the latency, bandwidth, processor capacity, memory, which are often ignored in the homogeneous systems where the assumption of uniformity is sound, must be considered. We designed HeRMeS, a tool allowing to perform simulations of adaptive applications in dynamic distributed environments. HeRMeS includes several general methods for dynamic load balancing and uses the system capabilities to compute and perform a new load redistribution. Online values for the system parameters are provided the Network Weather Service [5] tool (NWS), which periodically monitors and dynamically forecasts the performance of the network and of the computational resources over a given time interval.

3 Heterogeneous Computing Model

The execution time of an application dealing with unstructured meshes mapped onto a dynamic heterogeneous computing environment depends on two kinds of parameters: application specific and system specific. The actual mapping of the mesh subdomains onto processors induces a logical inter-processor communication topology. This can be denoted by a graph $G = (V, E)$, which is usually called subdomain graph. Its vertices correspond to the processors and the edges reflect computational data dependencies between data residing them. Let $V = \{1, \dots, p\}$ and $E = \{e_1, e_2, \dots, e_m\}$. Generally, this graph is assumed to be connected.

We assume theoretical heterogeneous computing model of type $H = (G, l, c, w)$ with the following significance of the components:

- G is the logical communication topology induced by the the application (we shall assume that only one subdomain is mapped to a processor);
- l is the vector of the processors' workloads, usually the number of vertices of each processor ;
- c denotes the p -dimensional vector of the capacities; without loss of generality one can assume that c is normalized, i.e. $\sum_{1 \leq i \leq p} c_i = 1$;
- w , is a m -dimensional vector of weights associated to the edges of G .

The load is fairly distributed in the system when

$$\frac{l_i}{c_i} = \frac{l_j}{c_j}, \forall i = 1, p.$$

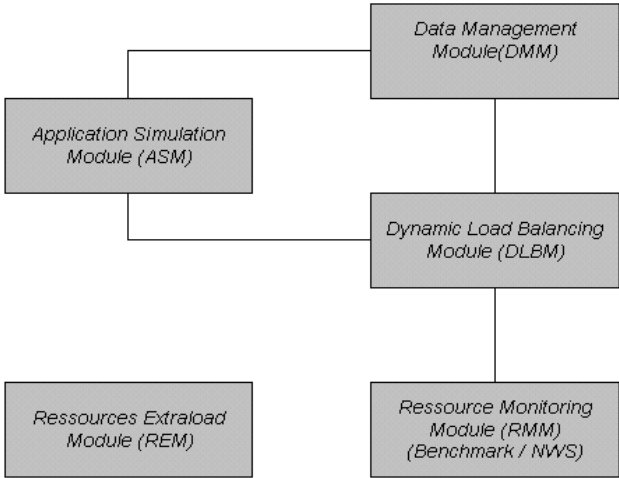
Relative to this model we designed, analyzed and developed dynamic redistribution techniques capable of assuring a fair execution of the concerned application [2, 4]. They generate a balancing flow on the edges of the communication graph G induced by the actual data distribution on the basis of the observed parameters. These methods minimize a certain distance, that constitutes an upper bound of the total cost of data migration.

4 Simulation Testbed

In order to be able to test the effectiveness of our dynamic load balancing techniques, a testbed simulating the behaviour of a generic parallel application performing adaptive computations was proposed. The goals of the simulation are:

- to test the effectiveness of the assumed theoretical model;
- to compare the proposed dynamic load redistribution techniques;
- to test the proposed algorithms in real situations;
- to identify the negative aspects and to improve the model.

The following software modules were designed and partly implemented: the *Application simulation module*, the *Data management module*, the *Dynamic load balancing module*, the *Resource monitoring module*. Additionally, an independent module simulating the activity of external users/processes was considered: *Resources extra-load module*.



The tasks of each module are summarized below:

Data management module. This module is in charge of the management of the unstructured mesh used by the application. An initial partition is computed using the MeTis [6] software package and mapped onto the processors. This module offers interface functions that prepares/fills the data structures (CSR format), distributes the data on processors, builds the communication graph, operates modifications on the structure of the mesh following the orders of the application simulation module (where to refine, which percentage), offer information about the mesh to the other modules.

Dynamic load balancing module. This module consists in a number of dynamic load balancing methods for a heterogeneous computing model. These methods are combined with a multilevel repartitioning scheme derived from the ParMetis [7] package. It requests information about the structure induced by the actual distribution. It chooses and applies a method for determining the amount of data to migrate and the destination, on the basis of the information about the system parameters (processor capacity, bandwidth, latency) and the application parameters (logical communication topology, size of subdomains, etc) that it receives from the other modules. The selection for migration is done according to a gain function which takes into account the physical characteristics of the inter-processor communication.

Resource monitoring module. It measures the processor related parameters by a benchmark. It also provides current values or forecasts of network or processor parameters (available capacity, latency, bandwidth, etc...) to the dynamic load balancing module. The module contains several interface functions which call NWS primitives. NWS furnishes the fraction of CPU available for new processes, the fraction of CPU available to a process that is already running, end-to-end TCP network latency, end-to-end TCP network bandwidth, free memory, and the amount of space unused on a disk. NWS is configured manually for the target system before conducting experiments. Typically, a sensor and a forecaster are started on each machine. The application process residing on each machine first requests from the data management module the neighboring machines in the communication graph induced by the data distribution w.r.t the application and then sends queries to NWS to extract values for the system parameters along the corresponding links. These are used for setting up values for the synthetic parameters in the assumed theoretical heterogeneous model which are then communicated to the dynamic load balancing module.

Application simulation module. It simulates the behaviour of an adaptive parallel solver. The general assumed scheme is that indicated in algorithm 1. Typically, there is a computational kernel that solves a system of equations (usually a sparse linear system). Here, one can identify a local computation part, local communication part for sending/receiving the boundaries between neighboring subdomains owned by the processors and a global communication part, mainly for evaluating the error after computing a certain norm. In a first step, we assume that the computation time of a processor

Table 1. Heterogeneous computing environment used for performing tests

| Machine name | OS | processor type | number | CPU | RAM |
|------------------|-------------|----------------|--------|---------|------|
| PC-151 .. PC-165 | Solaris 2.7 | i386 | 16 | 166 Mhz | 32 M |
| peridot | Solaris 2.7 | sparcv9 | 1 | 167 MHz | 64 M |
| jargon | Solaris 2.7 | sparc | 1 | 110 MHz | 64 M |
| outremer | Solaris 2.7 | sparc | 1 | 110 MHz | 64 M |
| zircon | Solaris 2.8 | sparc | 1 | 70 MHz | 32 M |
| tourmaline | Solaris 2.8 | sparc | 1 | 60 MHz | 32 M |
| girasol | Solaris 2.8 | sparc | 1 | 75 MHz | 32 M |

is a function of the number of nodes of the subdomain and the estimated speed of the processor. This is obtained by multiplying the processor speed with the fraction of available CPU provided by the Network Weather Service tool. The local communication time for exchanging the boundaries between two logically neighboring processors is a function of the edge-cut between the corresponding subdomains, the startup time for setting up a communication channel, the latency, the available bandwidth and the number of sent messages.

5 Experimental Setup

We used a cluster based on commodity hardware available in our institute. This consists in a collection of heterogeneous machines with the characteristics given in table 1. On these machines we have installed LAM-MPI ([8]) and Network Weather Service 2.0.6. At this time, the basic functionality of each module was implemented. In a first step, we focused our attention mainly on testing the inter-operability between the dynamic load balancing module and the resource monitoring module. A test suite was generated by using different load distribution and communication topologies furnished by the data management module. The resource monitoring module provided online values for network and processor parameters. The approach we used is a distributed one and the idea is the following: each processor knows what is its current load, who are its logical neighbors, its computing power measured by using an off-line benchmark. For finding out the forecasts for the available CPU fraction, latency and bandwidth on the links relying it with its neighbors, it sends requests to the resource monitoring module. It was tested the inter-operability between these modules and the data management module. The algorithms proposed for keeping the fairness at runtime were tested with subdomain graphs obtained from real test graphs used in CFD. The proposed dynamic load balancing methods were used with a multilevel scheme inspired by those provided by the ParMetis package. During the preliminary experiments performed, the weights w_i appearing in the synthetic model were set to the square of the inverse of the forecasted bandwidth along the

corresponding link. It should be noted that the differences are quite important and cannot be ignored. In our case it was observed that the bandwidth ranges between 3.5 and 17.8 and the normalized relative speeds between 0.036 to 0.120.

The testbed allows using a variety of dynamic load balancing algorithms, load distribution and logical communication topologies either artificially generated or corresponding to subdomain graphs of meshes usually used in engineering fields like CFD. Some preliminary tests were performed with several dynamic load balancing algorithms. The tables 2 and 3 report results in terms of execution times (expressed in seconds) and in terms of number of load index exchange phases for a highly imbalanced load distribution across the machines given in the table 1. The tested topologies were path, ring, star and subdomain graphs of some well known unstructured meshes (whitaker, airfoil, grid20x19 and shock). The first column corresponds to the algorithm described in [4] and the other 5 to several generalized diffusion algorithms described in [2]. For the last algorithms the tolerated convergence error used was 0.001

Table 2. Execution times for different communication topologies on the heterogeneous environment described in 1

| Topology | LPO | GD1 | GD2 | GD3 | GD4 | GD5 |
|--------------|------|-------|-------|-------|-------|--------|
| Path | 0.32 | 19.82 | 20.41 | 37.48 | 66.73 | 300.24 |
| Ring | 0.27 | 6.69 | 6.25 | 11.5 | 18.67 | 80.06 |
| Star | 0.20 | 8.79 | 8.96 | 9.60 | 9.93 | 4.00 |
| SD_whitaker | 0.24 | 2.40 | 2.70 | 5.90 | 10.46 | 13.14 |
| SD_airfoil | 0.23 | 3.55 | 6.02 | 9.70 | 17.27 | 20.45 |
| SD_grid20x19 | 0.20 | 2.05 | 2.90 | 4.96 | 5.80 | 7.11 |
| SD_shock | 0.23 | 3.97 | 6.85 | 8.66 | 11.21 | 11.82 |

Table 3. Number of load index exchange phases on the heterogeneous environment described in 1

| Topology | LPO | GD1 | GD2 | GD3 | GD4 | GD5 |
|--------------|-----|------|------|------|------|-------|
| Path | 21 | 1828 | 1950 | 3284 | 5678 | 23898 |
| Ring | 21 | 512 | 521 | 879 | 1537 | 6390 |
| Star | 21 | 961 | 983 | 995 | 1086 | 425 |
| SD_whitaker | 21 | 221 | 258 | 562 | 998 | 1183 |
| SD_airfoil | 21 | 387 | 650 | 1003 | 1638 | 1936 |
| SD_grid20x19 | 21 | 141 | 213 | 296 | 431 | 508 |
| SD_shock | 21 | 313 | 521 | 660 | 856 | 885 |

These tests provided us valuable information about the effectiveness of the employed redistribution techniques on a real heterogeneous environment and feed back of how to improve them.

6 Further Work

We are currently working on the full integration of the discussed modules. We are pursuing efforts towards reaching the proposed objective i.e. a tool simulating the behaviour of generic adaptive applications which offers dynamic load balancing support for the execution on real heterogeneous environments. Our dynamic load balancing methods can be applied incrementally when only the processing capacities or the loads vary and perform fastly when the communication topology or the network parameters vary dynamically. A broad number of experiments will be conducted, by varying the computational and communication requirements of the generic simulated application, in order to get a closer picture of how the system parameters influence the total execution time. A visualization module will be further added.

References

- [1] Jones, M. T., Plassman, P. E.: Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In Proceedings of the Scalable High Performance Computing Conference, IEEE Computer Society Press, pages 478-485, 1994. 88
- [2] Rotaru, T., Nägeli, H.-H.: The Generalized Diffusion Algorithm. Techn. Rep. RT-2000/06-1, Institut d'Informatique, Université de Neuchâtel, June 2000. 91, 94
- [3] Rotaru, T., Nägeli, H.-H.: Minimal Flow Generated by Heterogeneous Diffusion Schemes. In International Conference On Parallel and Distributed Computing and Systems, Anaheim, USA, August 21-24 2001.
- [4] Rotaru, T., Nägeli, H.-H.: A Fast Algorithm for Fair Dynamic Data Redistribution in Heterogeneous Environments. Submitted. 91, 94
- [5] Wolski, R., Spring, N., Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Technical Report CS98-599, 1998. 90
- [6] <http://www-users.cs.umn.edu/~karypis/metis/metis>. 92
- [7] <http://www-users.cs.umn.edu/~karypis/metis/parmetis>. 92
- [8] <http://www.lam-mpi.org>. 93

Simulating Cloth Free-Form Deformation with a Beowulf Cluster*

Conceição Freitas, Luís Dias, and Miguel Dias

ADETTI/UNIDE, Associação para o Desenvolvimento das Telecomunicações e Técnicas de
Informática, Edifício ISCTE, 1600-082 Lisboa, Portugal, www.adetti.iscte.pt
mc.freitas@oninet.pt,
{[luis.dias](mailto:luis.dias@adetti.iscte.pt),[miguel.dias](mailto:miguel.dias@adetti.iscte.pt)}@adetti.iscte.pt

Abstract. In this paper, we present the results of a parallel implementation of an algorithm which performs physical-based cloth free-form deformation, using a particle system modelling method. This sequential algorithm is characterised by being iterative over a large data set of coupled particles, which discretize cloth surface, and also of local nature in its mechanical simulation CPU-intensive method: the law of motion of each particle only depends in its local neighbourhood and is evaluated by solving an Ordinary Differential Equation (ODE). This characteristic makes the sequential algorithm suitable for a parallelisation. We have made a parallel algorithm implementation of the mentioned problem exploiting the computing capacity of a Beowulf Cluster. We also propose a simple load balancing method for this algorithm. The results show that substantial speedup gains can be achieved by dividing the particle system domain into several processes. To objectively evaluate the results obtained with the parallelisation technique, we have utilised known metrics related to measure the overall performance of the parallel algorithm such as Relative Speedup.

1 Introduction

Researchers from distinct areas have investigated the cloth-modelling problem by mathematical or physical methods since 1930 [5]. Since 1986 [10], the Computer Graphics community has published several works addressing this known problem. In the field of High Performance Computing distinct Parallel Systems have been developed to fulfil the need of increasing processing capacity, felt by the research community, as well as enterprises and organisations in the last decades. These systems can be classified as Distributed Processing Systems and Parallel Processing Systems [11]

In this paper, we focus our interest in the parallel simulation of cloth deformation using a Beowulf PC Cluster (or Beowulf). Beowulfs belong to the family of parallel computers architectures, known as clusters. Essentially, clusters comprise independ-

* This work is partially funded by the European Community IST Project Fashion-Me: Fashion Shopping with Individualized Avatars

ent and self-supporting systems that are networked to provide a means of communication [6]. In the Beowulf case, these systems are PCs available in the mass market and exhibiting excellent price/performance ratio and using widely available networking technology and open-source Unix type of operating systems [6, 7].

Other researchers have recently addressed this problem using shared memory parallel systems [8, 9].

2 Computing Fabric Free-Form Deformation

2.1 Sequential Algorithm

Our starting point is a sequential client-server computational framework (referred to as E-Fabric). In the E-Fabric, the server implements a computational model of plain-woven fabrics, based in the macroscopic Theory of Elasticity and in principles taken from the Mechanic of Materials. The model is able to represent known elastic behaviour in deformation, such as planar extension and shearing and out-of-plane bending, drape and Euler buckling [3]. Cloth is assumed to be an orthotropic linear elastic continuum, discretized by a mesh of triangles. Each triangle connects three particles and is able to quantify the stress and strain of the underlined medium. The equilibrium of the particle system is evaluated first by solving a linear system of equations for each triangle and then by solving the following ordinary differential equation (ODE), for each surface particle \mathbf{P}_i .

$$m_i \frac{d^2 \mathbf{r}}{dt^2} + w_d \frac{d\mathbf{r}}{dt} = \mathbf{c}(\mathbf{r}, t) + \mathbf{g}(\mathbf{r}, t) + \mathbf{f}^n(\mathbf{r}, t) \quad (1)$$

where m_i is the particle mass, w_d is a damping coefficient, $\mathbf{g}(\mathbf{r}, t)$ is the resultant of the internal tensile elastic forces of planar extension, compression and shearing, $\mathbf{c}(\mathbf{r}, t)$ is the resultant of the elastic force that resists bending and buckling deformation and $\mathbf{f}^n(\mathbf{r}, t)$ is the resultant of n external forces, such as gravity, impulsive contact forces, and wind force.

This equation is solved by using an adaptive step explicit method, with LSODES (*Livermore Solver for Ordinary Differential EquationS*)[13].

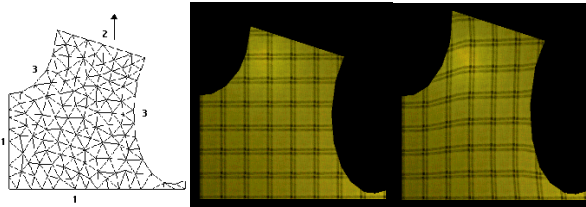


Fig. 1. Tensile deformation of part of a textile cutting pattern

The client can be configured to simulate seven different fabric deformation tests, valid for regular, of rectangular shape, or irregular (cutting patterns) fabric samples, including:

- 1 A tensile test with planar deformation along any direction in respect to the warp (simulating an Extensometer Tensile test, Figure 1);
- 2 Fabric draping with double bending (Figure 2).

The size and topology of the irregular fabric samples are defined by three output files of the *Triangle* program [12] which performs Delaunay triangularisation of the cutting patterns. These three files list the following fabric shape attributes: vertexes (or particle positions) of the interior and exterior contour, triangles and external edges.

In this version of E-Fabric there is no collision detection between moving particles and a rigid body. However, the cloth draping experiment (Figure 2) can be simulated using appropriate static boundary conditions in equation 1.

The client-server system was implemented in C++ and Open Inventor, in Silicon Graphics systems. Supported 3D geometric formats include VRML 1.0, VRML 2.0/97 and Open Inventor.

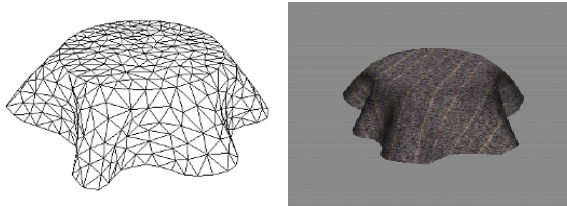


Fig. 2. Cloth draping simulation

2.2 Parallel Algorithm

To increase the computational efficiency and speed of the E-Fabric Server we have developed a parallel version of the E-Fabric server, which runs on a Beowulf Cluster.

The fundamental aim behind this transformation from sequential into parallel computing, is to specify and develop a software layer that is transparent to the E-Fabric Client, based in PVM version 3 (Parallel Virtual Machine) [1]. PVM is a software API that allows a network of heterogeneous Unix computers to be used as a single large parallel computer. PVM supports different applications, machines, and network levels. Our Beowulf Cluster computer architecture is depicted in Fig. 3.

Normally, the E-Fabric is used to simulate systems with several thousands triangles and hence an equivalent number of coupled particles, which makes this application a higher consumer of CPU. The amount of data to process is related directly with the CPU time, hence we have chosen initially the data parallelism model. This model is characterised by the application of the same operation, to multiple elements of the particle system state data structure, in concurrency (which includes, for each particle, its 3D position, velocity, acceleration, mass and boundary conditions). Our approach was to spawn not one, but several E-Fabric server processes in the Cluster, and to provide subsets of the initial particle system state data, for the various processes. The parent server process initialises the complete particle system data and its children, only a subset. In the end of each simulation time-step, one of the servers (the parent) consolidates the others (the children) partial computation results, and provides the new particle system state to the client, for visualisation.

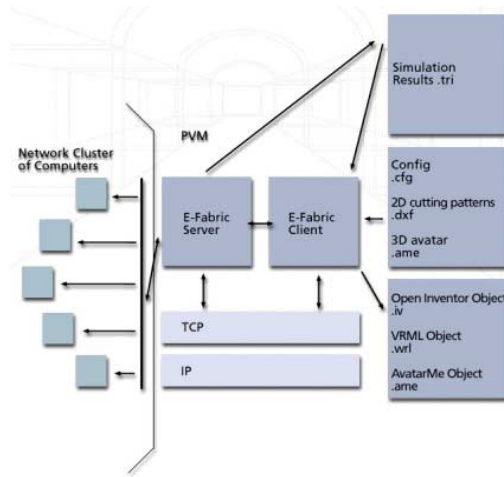


Fig. 3. Parallel Cluster Computer Architecture

To compute the subsets of the complete particle system, in the case of an irregular topology of triangles (which discretises the cloth surface), we have used the following algorithm: 1 Compute the mass centre of every particle triangle; 2 Sort the mass centres by x and by z ; 3 Evaluate the difference between maximum and minimum of x ; 4 - Evaluate the difference between maximum and minimum of z ; 5 - If the largest difference is along x we divide the data domain (particle triangles) along x , otherwise along z .

We begin to logically divide our program into several subprograms, each one corresponding to a process. The parent process initialises the whole particle system and then distributes the subsets of this particle system to the children processes.

In an initial approach, and for each time step of the simulation, each child computes a subset of the system and sends the results back to its neighbours and to its parent, via PVM messages. The parent computes its subset and sends the consolidated results to the client, for display.

Not all the processes execute the same code: the parent process initialises the whole particle system and sends to its children their subsets of the system; and then, the children initialise only a subset of the system. Hence, we can say this technique is a mixture of a data parallelism model and a functional model, where not all the processes execute the same operations.

During each time-step iteration, the children processes exchange border particle status information (position and velocity). To allow a more accurate determination of the various border particles states, we forced an overlap of neighbour process particle domains (or subsets of complete particle system data). This data domain overlapping procedure is made along the border of the domain partition. It consists in extending the data domain (a triangle mesh of particles) of each process with every particle's triangles of the neighbour processes that include the border particles (see Figure 4).

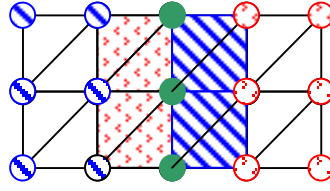







Fig. 4. This figure shows an example of the data domain overlapping scheme for two domains. The  symbol represents a particle of domain A,  corresponds to a particle of domain B and  is a border particle. Marked by  and  are particle triangles to be added to the domain A and domain B, respectively

Hence, when computing the border particle states, the forces resulting from the interaction with the particles belonging to the neighbour processes (which depend only in the particle state history) are accounted. As the parent process receives the border particle states twice, one for each neighbour worker process, the master process calculates an average for those particles position and velocity.

2.2.1 Load Balancing

In our load balancing method, the master process forces the worker process to stop the simulation only once, after a short specified time interval (the benchmark time). The workloads (or computational times) for each worker process are then determined and a repartition of the data domains is made. After this process the simulation is repeated from the first time step[1].

We assumed that the computational time of the algorithm is directly proportional to the number of the particles in the system (i.e. the algorithm is $O(n)$). This computational time is approximated as the difference between the elapsed time and total idle time. Therefore, at the benchmark time,

$$t_i \propto C_i \cdot p_i \quad (2)$$

where, for process i , t_i is the computational time, C_i is the proportional coefficient and p_i the ratio between the number of particles of the sub-system and the number of particles of the complete system.

The repartition of data domains is made to minimize the process idle time, which implies equality of all elapsed times after the repartition ($t_i = t_j$ for every i, j). Using this equality, substituting equation 2, and applying summations, we obtain the new data domain partition, for process r , p_r , depending only on the coefficients C_i , where N is the total number of processes.

$$p_r \propto \frac{1}{\sum_{i=1}^N C_i} \quad (3)$$

The master process computes the coefficients, C_i , at the benchmark time, using equation (2). This process then calculates the data domain repartition, using the above equation, and reassigns new data domains for each worker process accordingly.

3 Results and Discussion

We have executed our parallel algorithm for the case of tablecloth draping with double bending (see fig. 2). All our results refer to a circular fabric sample with 0,52 m diameter and a triangle mesh of 437 particles. These results are relative only to the parallel version of E-Fabric Server, which generates an output file *.trj* (see Figure 3). For visualization purposes we used the E-Fabric Client Program in batch mode, with the aid of the Open Inventor library.

Beowulf cluster has 4 LINUX Pentium II based computers linked by a switched Ethernet Network, running PVM3. The machines main specifications are: scluisa - Pentium II, 500 MHz, 256 Mbyte RAM, scana -Pentium II, 350 MHz, 128 Mbyte RAM, scines - Pentium II, 350 MHz, 64 Mbyte RAM, scmada: Pentium II, 350 MHz, 256 Mbyte RAM.

3.1 Relative Speedup Metric

We considered the relative speedup metric to measure the overall performance of the parallel algorithm [4]. The relative speedup $S_{relative}$ is the factor by which execution time of the initial algorithm is reduced on P processors. We define this metric as:

$$S_{relative} \square \frac{T_1}{T_p} \quad (4)$$

T_1 is the execution time of the uniprocessor and uniprocess sequential algorithm and T_p is the execution time of the parallel algorithm on P processors. The execution time of a parallel program is the elapsed time from where the first processor starts executing on the problem, to when the last processor completes execution.

To address the problem of the nodes being a little heterogeneous and the existence of several users of the cluster, we approximated T_p with the CPU time of the slowest processor and T_1 with the CPU time of the sequential algorithm running with this processor.

In figure 5 we depict the relation between the relative speedup and parent idle time and the number of hosts (or processors) in the Beowulf Cluster (the parent process is allocated to the scluisa node). The value corresponding to 1 host was the lower elapsed time for the sequential algorithm that we obtained for the various machines of the Cluster. The number of worker processes is equal to the number of nodes so that we could analyse the effect of increase of nodes in the performance of the algorithm (i.e. its scalability). We can observe a steady decrease in the parent idle time and an increase in the relative speedup to 2.8.

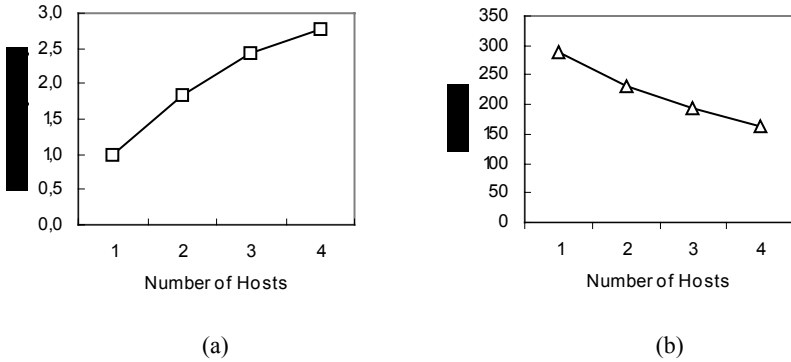


Fig. 5. Relative Speedup (a) and Parent Idle Time (b) vs number of hosts with the number of child processes equal to the number of hosts

3.2 Load Balance

To control the load imbalance in a cluster with 3 nodes and 3 worker processes, we ran an increased number of instances of the sequential algorithm in one selected node. To measure this load imbalance we used the metric $W_{\max} / W_{\text{avg}}$ where W_{\max} is the sum of the workloads (equal to computing times, see 2.2.1) (W_{comp}) on the most heavily-loaded processor, and W_{avg} is the average of load of all processors [2].

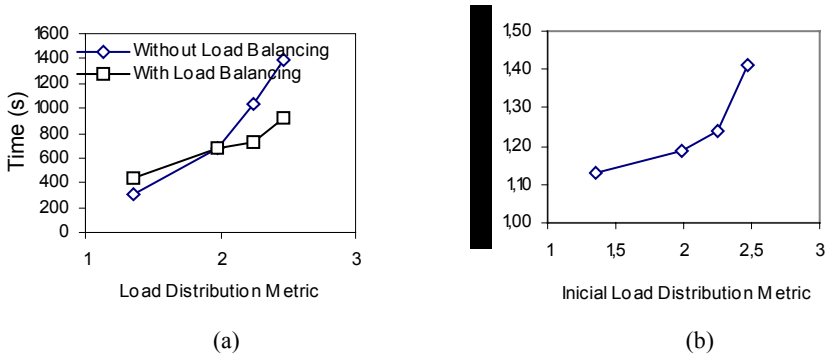


Fig. 6. Effect of load balancing in total execution time (cluster with 3 nodes and with the load variation in one node) (a) and in the load distribution metric (b)

In figure 6 we compare the results of the versions of the algorithm with load balancing and without our load balancing method. Looking at this results, one can conclude that our load balance strategy begins to 'pay-off' when load imbalance metric (or load distribution metric) is higher than 2. For the simulation with the most imbalanced workload the elapsed time decreased 33%. In this figure the final metric curve gives us the relation between the load imbalance metric before and after the one-time

domain repartition. It expected that more data domains repartitions (for this range of metric values) do not result in performance gains, as the final metric value after the domain repartition is always lower than the threshold value.

4 Conclusions

In this research work we have shown the improvement of the efficiency of cloth modelling, by developing a parallel version of a previously available physical-based algorithm [3].

From the comparison of the CPU time of a sequential program with that of its parallel equivalent, we have concluded, that the total observed enhanced performance of our parallel algorithm in a Beowulf Cluster, was more important than the individual performance of one node.

The results of the scalability of the algorithm are encouraging regarding its further deployment in Beowulf Clusters with more and faster nodes, so as to achieve real time cloth draping simulations.

We have concluded also, that a simple empiric load balancing method of the parallel algorithm, based on the benchmark of the processes workloads, is effective when the load imbalance metric is approximately higher than 2.

In the future, various aspects of this algorithm can be improved as follows:

- Use the knowledge of the processors computational profile and the estimation of the computational power required by sub-domains, to improve strategies for redistributing the workloads;
- Develop strategies for process recovery, when a processor or a process fails, because the PVM does not contain automatic recovery functions.

References

- [1] Al Geist, PVM 3 User's Guide and Reference Manual, 1994
- [2] Rupak Biswas, Leonid Oliker, Andrew Sohn, "Global Load Balancing with Parallel Mesh Adaption on Distribute-Memory Systems" SC96 Technical Papers, 1996
- [3] Dias, J. M. S., Gamito, M., N., Rebordão, J. M., A Discretized Linear Elastic Model for Cloth Buckling and Drape, Textile Research Journal 70 (4), pp 285-297, April 2000
- [4] Foster, Designing and Building Parallel Programs, 1995
- [5] Peirce, The Handle of Cloth as Measurable Quantity, 1930
- [6] Sterling, Thomas L. Sterling, Salmon, John, Becker, Donald J., Savarese, Daniel F., "How to Build a Beowulf, A Guide to the Implementation and Application of PC Clusters", The MIT Press, 1999.
- [7] Trezentos, P., "Projecto Vitará (M dulo1): Software de Comunica^{ção} entre Processadores" - DIPC/LAM/PVM, vitara.adetti.iscte.pt, 1999
- [8] Lario, R., Garcia C., Prieto, M. Tirado F., "Rapid Parallelization of a Multilevel Cloth Simulator Using OpenMP", Proc. International Conference on Parallel Architectures and Compilation Techniques, PACT'01, Barcelona, Spain, 2001
- [9] Romero, S., Romero, L.F., E.L.Zapata, Fast Cloth Simulation with Parallel Computers Proc. 6th Int'l Euro-Par Conference (Euro-Par 2000), Munich, Germany, 2000.

- [10] Feynman, Modeling the Appearance of Cloth, Cambridge, 1986
- [11] Foster, Designing and Building Parallel Programs, 1995
- [12] Shewchuk, J., R., Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, First Workshop on Applied Computational Geometry, ACM, 1996
- [13] Petzold, L. R. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations, Siam, J. Sci. Stat. Comput. (4), pp 136-148, 1983

Concept of a Problem Solving Environment for Flood Forecasting*

Ladislav Hluchy, Viet Dinh Tran, Ondrej Habala, Jan Astalos, Branislav Simo,
and David Froehlich

Institute of Informatics, SAS
Dubravská cesta 9, 842 37 Bratislava, Slovakia
hluchy.ui@savba.sk

Abstract. Flood forecasting is a complex problem that requires cooperation of many scientists in different areas. In this paper, the concept of a Collaborative Problem Solving Environment for Flood Forecasting – a part of the CrossGrid project – is presented. This paper also focuses on the parallel numerical solution of hydraulic simulation module that is one of the most computational-intensive parts of the whole system.

Keywords: Virtual Organization, Flood Forecasting, Collaborative Problem Solving Environment, Parallelization, MPI Programming.

1 Introduction

Over the past few years, floods have caused widespread damages throughout the world. Most of the continents were heavily threatened. Therefore, modeling and simulation of flood forecasting in order to predict and to make necessary prevention is very important. In this paper we propose to develop problem solving environment [1] meant as a support system for establishment and operation of Virtual Organization for Flood Forecasting (Chapter 2) associating a set of individuals and institutions involved in flood prevention and protection. The system will employ the Grid technology to seamlessly connect together the experts, data and computing resources needed for quick and correct flood management decisions. The main component of the system will be a highly automated early warning system based on hydro-meteorological (snowmelt) rainfall-runoff simulations. Moreover, the system will integrate some advanced communication techniques allowing the crisis management teams to consult the decisions with various experts. The experts will be able to run the simulations with changed parameters and analyze the impact (what-if analysis). The

* This work is supported by EU 5FP CROSSGRID IST-2001-32243, ANFAS IST-1999-11676 RTD projects and the Slovak Scientific Grant Agency within Research Project No. 2/7186/20.

Parsons Brinckerhoff Quade and Douglas, Inc., 909 Aviation Parkway, Suite 1500, Morrisville, North Carolina 27560 USA, Froehlich@pbworld.com.

use of Grid resources is vital especially in the case of flood crisis when the simulations have to be performed as fast as possible.

In recent years, since the 1991 workshop on Problem Solving Environments, held in Washington, a lot of research in this area has been done. PSEs are no more used exclusively to solve partial differential equations. They are used as expert systems for a wide scale of science applications, from environmental sciences [2] through biology (and even DNA mapping [13]) to military problems [18]. Of course, the main stream is oriented towards mathematical tools, computer-aided design and high-energy physics simulations.

PSE brings together computer resources, virtual libraries, databases, visualization tools and expert systems. Current trends present to us a new way of cooperative problem solving - Collaborative PSE, using the Internet technology to connect geographically separated researchers. In this paper, we describe a Collaborative Problem Solving Environment for Flood Forecasting. Flood forecasting, when done precisely, is a computationally intensive problem, requiring considerable resources. Therefore, behind the user-friendly interface to the PSE, Grid technology will be used to run complex weather and water-flow simulations.

2 Virtual Organization for Flood Forecasting

As said before, flood forecasting is a computationally intensive process. To gain the resources needed for it, we have proposed creation of a virtual organization with the following entities involved:

- Data providers - these organizations provide the actual physical measurements of real weather conditions, upon which the predictive simulations are based.
- Storage providers - these organizations will provide storage space, archives and databases to store all data needed in the CPSE. The data storage process will not be a trivial one. The system will need to store a huge amount of measured data, pre-simulated scenarios used for rapid situation evaluation, simulation configurations and terrain maps.
- Cycle providers - these organizations will provide the computational resources for the simulations. It will be a group of cluster maintainers, organizations owning supercomputers and pools of workstations.
- Portal - this is the gate to the whole PSE and a core of its management.
- Users - the organizations that will profit from the results of the simulations run using the resources provided by the rest of the virtual organization.

As shown in Fig. 1, the whole system will use the Grid Infrastructure to connect individual parts together. We will use the Globus toolkit. It can provide means of secure authentication of persons accessing the PSE, as well as VO participants cooperating on completion of some task (e.g. a cycle provider running some simulation and for its sake accessing a remote database with input data, managed by another VO participant). The Globus toolkit provides cryptographically secure authentication methods, based on the X.509 standard. Furthermore, it provides LDAP services via Globus MDS (Metacomputing Directory Services) package, which can be

used to create an elaborated information infrastructure describing the resources in the VO, together with their static and dynamic attributes.

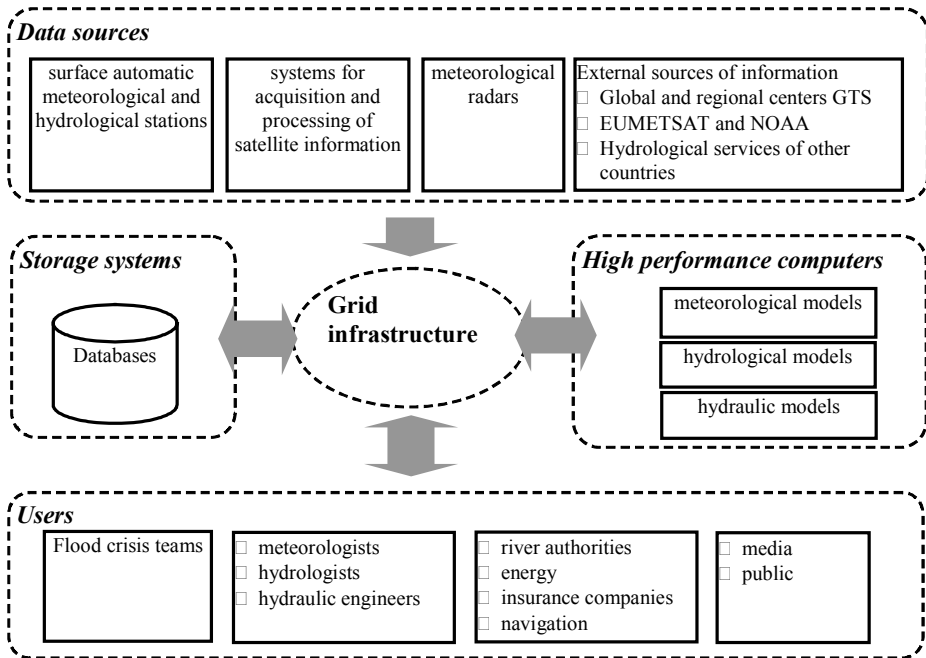


Fig. 1. Virtual Organization for Flood Forecasting

The process of weather forecasting relies on a stream of measured input data, which consists of:

- data provided by surface automatic and semiautomatic meteorological and hydrological stations, measuring for example precipitation, wind speed and direction, river depth or snowpack;
- data provided by satellite imagery; data provided by meteorological radars, mainly now-casting of precipitations;
- external sources of information, like GTS centers, EUMETSAT or foreign hydrological services.

In the normal operational mode, the system will be used for common meteorological and/or hydrological forecasting. Meteorological forecasts will be computed (according to common habits of SHMI) twice a day, with the possibility to issue more simulation runs based on more accurate and actual data, when needed. Based on the results of the meteorological simulation, hydrological models may be (and will be) started to predict river status. The last step of the cascaded forecasting - hydraulic simulation - will be probably used only in the case of a substantial increase in water level of the target flow, when there is higher possibility of a flood. The data gained by running common meteorological model, which will be the ALADIN model, will be evaluated in order to assess possible local flash flood danger. When there is

such a danger, another, more accurate and detailed (and also more demanding) model will be used on the locality threatened by the flash flood. When an exceptional event like a flood or a flash flood is about to occur, the system will concentrate its resources on the solution of it, suppressing all common activities described in the previous paragraph. The users-experts will need to run simulations for a set of possible weather evolutions, and also the pre-calculated scenarios will be used. In Fig. 1, there is a user named "Flood crisis team", which is actually a team of environmental experts created in order to quickly react to the oncoming situation.

3 Cascade of Simulations

Flood forecasting requires quantitative precipitation forecasts based on the meteorological simulations of different resolution from meso-scale to storm-scale. Especially for flash floods, high-resolution (1km) regional atmospheric models have to be used along with remote sensing data (satellite, radar). From the quantitative precipitation forecast hydrological models are used to determine the discharge from the affected area. Based on this information hydraulic models simulate flow through various river structures to predict the impact of the flood. The results of simulations can be interactively reviewed by experts using the PSE, some of them accepted and forwarded to the next step, some rejected or re-run with modified inputs.

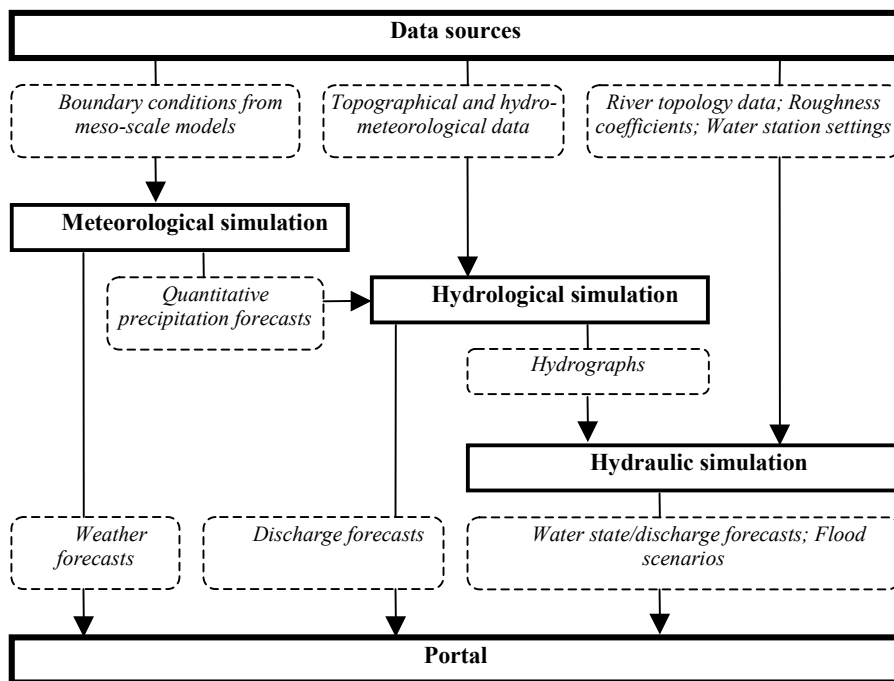


Fig. 2. Cascaded simulation scheme

So the cascade of simulations, which we have just described, starts by running a meteorological model. The plan is to use several of them, but the primary one will be the ALADIN model, currently used in SHMI. This model has been modified so that it can be used in a cluster environment, with MPI as a messaging interface. The model will be run twice a day, creating forecasts for the next 48 hours. The forecasts will consist of precipitation, wind speed and direction in several heights.

The output from the ALADIN model will be used to run hydrological simulations. Here we will again use more models, some simple and some more elaborated (and more demanding). These models will be run in a high-throughput environment, using a pool of workstations and the Condor management toolkit. The output from these simulations will be a hydrograph, representing estimated water level in the target area. This hydrograph will be used in the last stage of the simulation cascade, in the hydraulic simulation (Fig. 2).

4 Hydraulic Simulation at the Vah River Pilot Site

As meteorological simulation was already parallelized and hydrological simulation needs high-throughput facilities, in this paper we have focused on the parallelization of hydraulic simulation and applied it for actual river.

Vah river is one of the biggest rivers in Slovakia, therefore it was chosen as a pilot site for the project. Modeling and simulation of flood need following input data:

- Topographical data: There are several sources of input data for topographical data: orthophotomaps, river cross-sections, LIDAR (Light Detection and Ranging) data. GIS (Geographical Information System) is used to manage, combine these data and produce a final map for simulation (e.g. orthophotomaps are better for identifying objects, LIDAR cannot measure river depths).
- Roughness conditions: These data cannot be directly obtained, but are derived from orthophotomaps.
- Hydrological data: These data are obtained from monitoring past flood events. They can be used as boundary conditions as well as for calibration and validation of models.

Creating and calibrating models is a challenging process that can be done only by experts in hydrology. SMS (Surface-water Modeling System) [3] provides a convenient graphical interface for pre- and post-processing that can help the experts speedup the modeling process. The main hydraulic modeling module is FESWMS (Finite Element Surface-Water Modeling System) Flo2DH [12], which is a 2D hydrodynamic, depth averaged, free surface, finite element model supported by SMS.

5 Parallelizing Flo2DH

Simulation of floods based on Flo2DH is very computation-expensive. Several days of CPU-time may be needed to simulate large areas. For critical situations, e.g. when

an oncoming flood is simulated in order to predict which areas will be threatened, and to make necessary prevention, long computation times are unacceptable. Therefore, the use of HPCN platforms to reduce the computational time of flood simulation is imperative. The HPCN version of FESWMS Flo2DH not only reduces computation times but also allows users to exploit computational power available in Grid for simulation of large problems, and consequently provides more reliable results. The Grid infrastructure needed for running Flo2DH simulation on remote high-performance computers via portals (Fig. 1) is being developed.

Solving the systems of linear equations is the most time- and memory-consuming part of Flo2DH. It is also the part of Flo2DH that is most difficult to parallelize because the matrices generated by Galerkin FEM are large, sparse and unstructured. Galerkin FEM method has small computation time in comparison with linear solvers and it is trivially parallelized by domain decomposition. Therefore, the following part of this paper focuses on parallel linear solvers.

In order to achieve better performance, iterative linear solvers based on Krylov subspace methods [16] are used in parallel version of Flo2DH instead of the direct solver. These iterative solvers consist of matrix and vector operations only; thus, they offer large potential parallelism. In comparison with the direct solvers, iterative solvers are often faster and require less additional memory. The only drawback of iterative solvers is that they do not guarantee convergence to a solution. Therefore, preconditioning algorithms [17] are used to improve the convergence of iterative algorithms. Recent studies [4], [7] show that the combination of inexact Newton iteration [6] with Krylov methods and additive Schwarz preconditioners [14] can offer high performance and stable convergence. PETSC [9] library is used in implementation of parallel Flo2DH version.

6 Case Study

Input data are generated by GIS which combine the data from LIDAR, cross-sections and orthophotomaps. Hydrological data are provided by the Vah River Authority which monitors and records all past events in the Vah river. Roughness is defined on the basis of image processing procedures of orthophotomaps which divided the simulated pilot site to areas with the same coverage.

Hardware Platform

Experiments have been carried out on a cluster of workstations at the Institute of Informatics. The cluster consists of seven computational nodes, each of which has a Pentium III 550 MHz processor and 384 MB RAM, and a master node with dual Pentium III 550 MHz processors and 512 MB RAM. All of the nodes are connected by an Ethernet 100Mb/s network. The Linux operating system is used on the cluster. Communication among processes is done via MPI [8].

Experimental Results

The steady-state simulation converged after 17 Newton iterations. The original Flo2DH with a direct solver takes 2801 seconds for the simulation. The parallel version of Flo2DH takes 404 seconds on a single processor and 109 seconds on 7 processors (Tab. 1). One of the reasons why the parallel version (with iterative solvers) is much faster than original version (with a direct solver) is that the direct solver in the original version needs a lot of additional memory for saving LU triangular matrices. As there is not enough physical memory for the matrices, a part of them have to be saved on hard disk, which causes performance penalty. Speedup of the parallel version on our cluster is shown in Fig. 3.

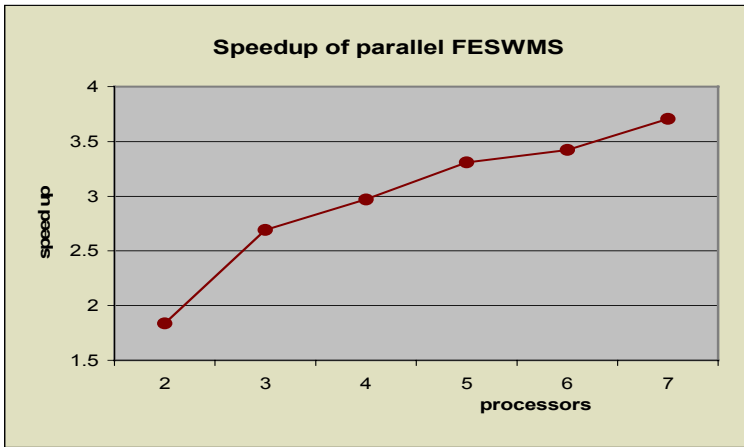


Fig. 3. Speedup of the parallel version of FESWMS

Table 1. Simulation times (in seconds)

| Original version | Parallel version | | | | | | |
|------------------|------------------|------|------|------|------|------|------|
| | 1 processor | 2 p. | 3 p. | 4 p. | 5 p. | 6 p. | 7 p. |
| 2801 | 404 | 220 | 150 | 136 | 122 | 118 | 109 |

From the experiments, among implemented Krylov iterative algorithms in PESTC, Bi-CGSTAB [10] is the fastest and relatively stable, especially when it is used with additive Schwarz/ILU preconditioner. GMRES [5] has the most stable convergence, however it is much slower than Bi-CGSTAB, and CGS [11] has unstable convergence.

7 Conclusion

In this paper, we have proposed a collaborative problem solving environment and a virtual organization for flood forecasting being developed in the scope of CrossGrid

project. The main component of the system will be a highly automated early warning system based on cascaded simulation of meteorological, hydrological and hydraulic simulation. The paper was also focused on parallelization of hydraulic simulation module Flo2DH, because this module was not satisfactorily parallelized. Experiments with real data from the Vah river pilot site show good speedups of the parallel version of Flo2DH. Similar results are achieved also on real data from the Loire river pilot site, France. Experiments on larger computer systems (Origin 2800 with 128 processors and Linux clusters with 200 nodes) are planned.

References

- [1] Gallopoulos, S., Houstis, E., Rice, J.: Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. IEEE Computational Science and Engineering Magazine, 1994, Vol. 2, pp. 11-23.
- [2] Fine, S. S., Ambrosiano, J.: The Environmental Decision Support System: Overview and air quality application. American Meteorological Society: Symposium on Environmental Applications, pp. 152-157. 1996, Atlanta, USA.
- [3] SMS: 2D surface water modeling package
http://www.bossintl.com/html/sms_overview.html
- [4] I.S. Duff, H.A. van der Vorst: Developments and Trends in the Parallel Solution of Linear Systems, Parallel Computing, Vol 25 (13-14), pp.1931-1970, 1999.
- [5] Y. Saad, M. H. Schultz: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Scientific and Statistical computing, Vol. 7, pp. 856-869, 1986.
- [6] R.S. Dembo, S.C. Eisenstat, T. Steihaug: Inexact Newton methods. SIAM J. Numerical Analysis, Vol. 19, pp. 400-408, 1982.
- [7] W.D. Gropp, D.E. Keyes, L.C. McInnes, M.D. Tidriri: Globalized Newton-Krylov-Schwarz Algorithms and Software for Parallel Implicit CFD. Int. J. High Performance Computing Applications, Vol. 14, pp. 102-136, 2000.
- [8] MPICH - A Portable Implementation of MPI
<http://www-unix.mcs.anl.gov/mpi/mpich/>
- [9] PETSc The Portable, Extensible Toolkit for Scientific Computation.
<http://www-fp.mcs.anl.gov/petsc/>
- [10] H. Vorst: Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. SIAM J. Scientific and Statistical Computing, Vol. 13, pp. 631-644, 1992.
- [11] P. Sonneveld: CGS, a fast Lanczos-type solver for nonsymmetric linear systems. SIAM J. Scientific and Statistical Computing, no. 10, pp. 36-52, 1989.
- [12] FESWMS - Finite Element Surface Water Modeling.
<http://www.bossintl.com/html/feswms.html>
- [13] Zibelin, D., Parmentier, T.: Distributed Problem Solving Environment Dedicated to DNA Sequence Annotation. Knowledge Acquisition, Modelling and Management, pp. 243-258, 1999.

- [14] X.C. Cai, M. Sarkis: A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Scientific Computing* Vol. 21, pp. 792-797, 1999.
- [15] D. Froehlich: Finite element surface-water modeling system: Two-dimensional flow in a horizontal plane. User manual.
- [16] Y. Saad, H. Vorst: Iterative Solution of Linear Systems in the 20-th Century. *J. Comp. Appl. Math*, Vol. 123, pp. 1-33, 2000.
- [17] K. Ajmani, W.F. Ng, M.S. Liou: Preconditioned conjugate gradient methods for the Navier-Stokes equations. *J. Computaional Physics*, Vol. 110, pp. 68-81, 1994.
- [18] Fox, G., Furmanski, W.: Problem Solving Environments from Simulation, Medicine and Defense using the Web. CRPS Annual Meeting, May 1996. <http://www.npac.sur.edu/users/gcf/crpcsemay96/fullhtml.html>

A Comprehensive Electric Field Simulation Environment on Top of SCI

Carsten Trinitis, Martin Schulz, and Wolfgang Karl

Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Technische Universität München (TUM), Institut für Informatik
Arcisstr. 21, D-80290 München

Tel: +49-89-289-{25771,28399,28278}, Fax: +49-89-289-28232
{Carsten.Trinitis, Martin.Schulz, Wolfgang.Karl}@in.tum.de
<http://wwwbode.in.tum.de/>

Abstract. A key aspect in the design process of high voltage gear is the exact simulation of the electrostatic and/or electromagnetic field distribution for three dimensional problems. However, such simulation runs are quite compute- and communication-intensive. Therefore, clusters of commodity PCs, equipped with high-speed interconnection technologies, are becoming increasingly important as target platforms, due to their excellent cost-performance ratio.

This paper gives a status report on the results that were obtained on such a Linux-based cluster platform (connected via Fast Ethernet and SCI) installed at ABB Corporate Research using POLOPT, a state-of-the-art parallel simulation environment for electrostatic and electromagnetic three dimensional problems from High Voltage Engineering. For electrostatic and electromagnetic field simulations in practical High Voltage Engineering, high efficiency was obtained. In addition, a unified execution environment based on MPI has been developed for the POLOPT environment.

Keywords: Electric field simulation, Commodity clusters, Parallel Efficiency, High voltage engineering, SCI

1 Motivation

One of the most important tasks in electrical engineering, especially for the design of high voltage gear like transformers, switchgear, or insulators is the precise simulation of the electrostatic and/or electromagnetic field distribution in areas of interest. A well established method for carrying out such simulations is the Boundary-Element method (BEM) [2]. By knowing the electrostatic field distribution possible flashovers in critical areas of the device can be avoided. Furthermore, eddy currents, a well known problem in the design of transformers, can lead to power losses and heating.

To solve the numerical equation systems resulting from the electrical boundary conditions, iterative solution techniques such as GMRES [10] are employed. For the electrostatic problem formulation, first experiences with a PVM-based version of the three dimensional electric field simulation program POLOPT [1] on a heterogeneous cluster of UNIX workstations have been presented in [3]. The PVM based electrostatic solver has then been ported to a cluster of LINUX based PCs [4]. To be able to use this PVM based electrostatic solver on top of SCI (for which an optimized MPI version is available), a PVM-MPI wrapper has been developed which can be inserted as an additional layer between MPI and the application.

For electromagnetic problems, a special solver has been developed by the Seminar for Applied Mathematics at the ETH Zürich, Switzerland [5]. This MPI-based electromagnetic POLOPT solver [5], [11], [12] has also been implemented on a LINUX-based PC cluster. Since the algorithm is rather communication intensive, a high bandwidth and low latency network like SCI [8, 6] is an important aspect.

Together, these two codes form a comprehensive simulation system on top of a single execution environment based on MPI. This allows to reduce the overall system complexity and the associated system administration cost.

2 The Simulation Process

To determine possible spots for flashovers or losses caused by eddy currents in transformers, comprehensive simulation calculations are necessary. The simulation software POLOPT offers modules for the required calculations based on the Boundary Element Method (BEM) [2].

Before the actual simulation can be started, a CAD model of the problem must be generated and the boundary conditions (i.e. voltages, material properties etc.) must be assigned. This is usually done with a commercial CAD package.

When the actual solver is applied to the problem, a coefficient matrix is generated by POLOPT which can be done in parallel since the generation of a matrix row is independent from all other lines. Depending on the problem type (electrostatic or electromagnetic), the resulting equation system is solved or a parallel preconditioner and solver (described in [5]) are applied.

The size of the equation systems are usually in the range of $10sup3$ to $10sup5$ unknowns with densely populated coefficient matrices.

The electrostatic and the electromagnetic solver have been parallelized on a LINUX based PC cluster installed at ABB Corporate Research, Heidelberg/Ladenburg, Germany.

3 Exploiting SAN-based Clusters

Clusters built from commodity components form ideal platforms for applications like the one discussed here. They are inexpensive while providing state-of-the-art

single node performance and hence exhibit an exceptional price/performance ratio. Together with appropriate freeware implementations of widely used parallel libraries, like PVM or MPI, they provide easy-to-use and portable platforms.

3.1 Clustering with SCI

A large group of applications, however, suffers from the low communication performance of commodity clusters, as they rely on general networking hardware, like Ethernet, and heavy protocol stacks, like TCP/IP or UDP/IP. To compensate for this deficit, so-called System Area Networks (SANs) have been developed which offer both high bandwidth through novel interconnection hardware and low-latency by relying on user-level communication schemes. The latter technique enables applications to directly talk to the networking hardware avoiding any operating system, protocol, or driver overhead.

One well-known example for this kind of technology is the Scalable Coherent Interface (SCI) [6], [8]. It is based on a global physical address space allowing each node to directly address any physical memory location within the system using standard user-level read and write operations¹. This leads to one-way latencies of under $2\ \mu\text{s}$ and a bandwidth of up to 320 MB/s (when connected via PCI busses with 64bit and 66MHz).

In order to harvest these capabilities of SCI and make them accessible to applications without significant overhead, a comprehensive and specially optimized software infrastructure is required. Such a framework has been developed in the SMiLE (Shared Memory in a LAN-like Environment) project [9]. It contains optimized libraries for all major parallel programming models from both message passing and shared memory which directly layer on top of the SCI hardware and are hence capable of fully exploiting them.

3.2 Unified Execution Environment

Despite the existence of such a multi-paradigm software infrastructure, supporting a setup with applications based on several different programming models is cumbersome and expensive to maintain. This was also the case for the POLOPT environment since the electric solver was originally based on PVM, while the newer electromagnetic solver is built on top of MPI.

This leads to a scenario in which a simulation environment is required to support both libraries and their respective execution environments. These are actually quite different: While MPI relies on a static node configuration file and initiates any computation on these nodes at application startup, PVM relies on a running virtual machine across all participating nodes which needs to be setup in advance. This heterogeneity leads to two separate versions of startup scripts and additionally raises questions with respect to configuration consistency.

In addition, the PVM libraries are normally associated with a higher overhead compared to MPI. This stems from extra functionality included in PVM with

¹ More on the principle of SCI can be found in [6].

the aim to support heterogeneous node configurations. Such support, however, is unnecessary in cluster environments discussed here which are built from standard PCs and hence are architecturally homogeneous. As this problem is inherent to PVM's approach, an optimized port to SCI taking advantage of its enhanced communication features does also not solve the problem [4].

In order to reduce these overheads and to prevent the problems connected with having to support multiple execution environments, it would be preferable to only run codes in an MPI environment. This would fully exploit the communication potential in homogeneous environments, reduce the number of software layers that need a special SCI adaptation, and enables the use of a single set of execution scripts.

3.3 Substituting PVM with MPI

Following these ideas, efforts have been undertaken to move the electrostatic simulation from PVM to an MPI base. In order to avoid code changes in the application itself as far as possible, this has been done using an application independent wrapper, mapping each PVM call to a corresponding implementation using MPI. Many routines can thereby be mapped one to one, as MPI and PVM both provide semantically similar routines. This e.g. applies to the straightforward blocking send and receive routines. Others have to be emulated by several MPI calls, like the non-blocking receive, which exists in PVM as a single routine, while MPI requires a probe operation followed by a blocking receive.

Different treatment, however, is required for the buffer management. PVM relies on explicit pack and unpack operations, which were introduced to support data conversion in heterogeneous architectures. Even though this feature is not needed anymore in MPI environments, the PVM API semantics needs to be maintained, leading to the implementation of a corresponding buffer management also in the PVM wrapper. For this a dynamically resized buffer is maintained by the wrapper and used for temporary storage of outgoing and incoming messages.

In addition to these communication related routines, startup and closing procedures require special treatment due to the modified execution environment. While MPI relies on a static processor allocation, PVM offers a dynamically changing virtual machine. This is, however, rarely used to its full extend in typical HPC applications, as they mostly rely either on SPMD-style or master/slave communication with a fixed number of tasks. On the contrary, it is even counterproductive in some space sharing scenarios, as the number of task used for a computation are spawned by the application itself and hence can not be determined a priori. This, however, is necessary to implement an efficient resource management and job allocation scheme. The wrapper therefore omits the dynamic scheme and provides mechanisms to map the PVM code to this static scheme. This is the only part in which the application itself had to be altered at a few, very limited locations in order to guarantee a safe startup.

Following this approach, the PVM-based parts of POLOPT have been converted into an MPI application without major efforts and resulting in less run-

time overhead in the communication layer. In addition, this approach leads to a unification of the runtime environments, as both codes now internally run on MPI and not anymore on their specific environment. This greatly simplifies the use of the code and reduces system administration costs.

This wrapper, however, can be used beyond the scope of POLOPT as it does principally not rely on code specifics. The only prerequisites are that the target code relies on a static task model that is intended for architecture-homogeneous clusters. It can therefore help to port a large range of PVM codes to MPI without major efforts and without requiring significant code changes. Especially the latter one is of high importance, as these kind of codes often fall into the category of so-called dusty decks applications. For those codes normally only very little documentation is available making any porting efforts within the application a risky and error-prone task.

3.4 Production Setup at ABB

The ABB production cluster used for our simulations consists of eight single-CPU Pentium-III class PCs, each equipped with 1 GB of physical memory. The nodes are connected by both Fast Ethernet and SCI, where the latter one is set up as a 2-dimensional torus in a 4x2 configuration.

All nodes run a standard installation of Linux Redhat using a 2.2 kernel, enhanced by some tools for cluster configuration and control. The MPI implementation used is either MP-MPICH 1.3 for Ethernet communication or ScaMPI [7] for SCI. The latter is a commercial MPI version specifically tailored and optimized for SCI-based clusters. It provides half round-trip latencies of zero length MPI messages of less than 5 μ s and a sustained bandwidth between 80MB/s (using the 32bit, 33MHz busses) and 312 MB/s (using 64bit, 66MHz busses). Both MPI versions provide the same API capabilities and a very similar execution environment, allowing a seamless transition between the two MPI implementations.

Besides this software environment installed on the cluster itself, the setup at ABB also includes a sophisticated job submission system allowing outside users, sitting in the development departments of ABB, to create and submit calculation jobs for the production cluster. This system is built in JAVA in a client/server fashion and includes a comprehensive user-side framework integrated into the CAD design process for high-voltage gear. In addition, this framework includes support for authentication and authorization, providing a safe and reliable way to share the capabilities of the cluster.

4 Electrostatic and Electromagnetic Examples and Results

Two real world problems have been simulated for the measurements described in this work. The first model, an electrostatic problem, is depicted in Figure 1. It is an SF_6 gas insulated loadbreak as being used for power supply systems by ABB

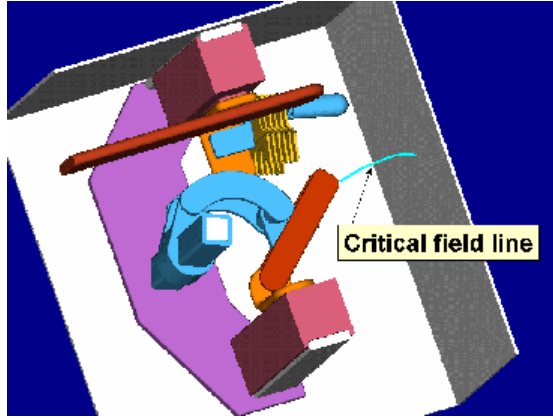


Fig. 1. Loadbreak Model

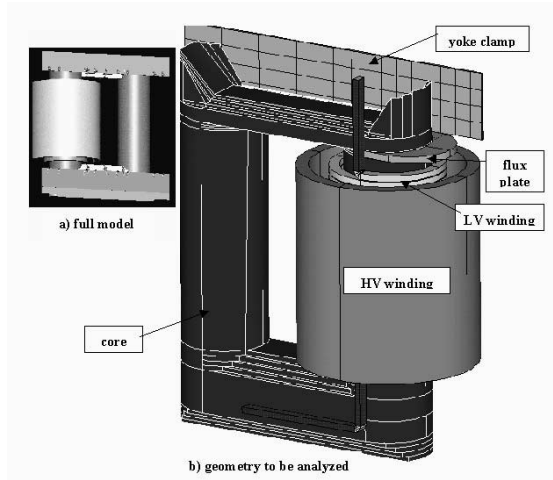
Norway. The simulation is carried out in order to determine possible hot spots for the electric field which can lead to flashovers in prototypes. The equation system has a dimension of about 15000 unknowns with a fully populated matrix. Using the PVM–MPI wrapper described in section 3, it was possible for the first time to run electrostatic problems on top of MPI.

The second example, which is described in more detail in [11], is a transformer model, as depicted in Figure 2. The problem is to determine the loss distribution in the yoke clamping plates to detect possible temperature hot spots. The model consisting of the magnetic core, the yoke clamps, the flux plate and the HV windings has been simulated by calculating the eddy currents in the yoke clamp. The problem (1300 unknowns, complex double) has been analyzed with respect to calculation times and parallel efficiency. For this example, the electromagnetic solver described in [5] has been utilized.

Simulation times for the electrostatic model can be seen in Table 1. They show an excellent scalability of the code with a parallel efficiency of 89% on 8 nodes for the overall computation time. Even a slight superlinear speedup was achieved due to increased aggregated memory bandwidth. However, it must be noted that using SCI in this case provided only a marginal benefit, as the electrostatic code is not very communication intensive and hence can not fully exploit the increased performance in the underlying network. However, it has been integrated into the unified execution environment by utilizing the PVM–MPI wrapper.

For electromagnetic problems, the results are shown in Table 2. These problems had to be calculated on a minimum number of 2 nodes due to the fact that the coefficient matrix had to fit into main memory. Therefore, the parallel efficiency has been measured relative to the computation times for two nodes.

The part of the simulation with the most communication between the nodes is the solver phase. As can be seen from the computation times for solving the

**Fig. 2.** Transformer Model**Table 1.** Computation times in seconds for full loadbreak problem

| Network | Seq. | FE. | FE. | SCI | SCI |
|---------|------|-----|-----|-----|-----|
| Nodes | 1 | 4 | 8 | 4 | 8 |
| Solve | 1139 | 279 | 148 | 273 | 144 |

Table 2. Computation times in seconds for transformer problem

| Network | FE. | FE. | FE. | SCI | SCI | SCI |
|---------|-----|-----|-----|-----|-----|-----|
| Nodes | 2 | 4 | 8 | 2 | 4 | 8 |
| Solve | 387 | 304 | 241 | 365 | 185 | 96 |

equation system, for Fast Ethernet a parallel efficiency of 64% (4 nodes) to 40% (8 nodes) compared to 98.7% (4 nodes) to 95% (8 nodes) on SCI can be obtained for the transformer model. Here, the use of SCI yields a significant improvement in parallel efficiency as the electromagnetic simulation is communication intensive and hence relies on a fast interconnection medium in order to achieve optimal scalability.

5 Conclusion and Outlook

The Scalable Coherent Interface (SCI) has been successfully integrated into the ABB POLOPT simulation environment for both PVM and MPI based codes. This has been accomplished by developing a PVM–MPI wrapper which allows a unified execution environment for both electrostatic and electromagnetic problems. For electrostatic problems, this allows a clean integration of the PVM code

base into the existing MPI production environment. Performance measurements with the electrostatic module have shown that the parallel efficiency is very good for both Fast Ethernet and SCI based networks. For electromagnetic problems, however, using the SCI network showed a significant performance gain for the solver, due to the fact that the algorithm for solving this equation system is rather communication intensive.

References

- [1] Z. Andjelic. *POLOPT 4.5 User's Guide*. Asea Brown Boveri Corporate Research, Heidelberg, 1996. 115
- [2] R. Bausinger and G. Kuhn. *Die Boundary-Element Methode*. Expert Verlag, Ehingen, 1987. 114, 115
- [3] A. Blaszczyk and C. Trinitis. *Experience with PVM in an Industrial Environment*. Lecture notes in Computer Science 1156, EuroPVM'96, Springer Verlag, pp. 174-179, 1996. 115
- [4] M. Eberl, W. Karl, C. Trinitis, and A. Blaszczyk. Parallel Computing on PC Clusters — An Alternative to Supercomputers for Industrial Applications. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain*, volume 1697 of *LNCS*, pages 493–498. Springer Verlag, Berlin, September 1999. 115, 117
- [5] G. Schmidlin, Ch. Schwab et al. *Preconditioning second kind Boundary Integral Equations for 3-D Eddy Current Problems*. ETH Zurich, Switzerland, 2000. 115, 119
- [6] 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. 115, 116, 121
- [7] L. Huse, K. Omang, H. Bugge, H. Ry, A. Haugsdal, and E. Rustad. *ScaMPI — Design and Implementation*, chapter 14, pages 249–261. Volume 1734 of Hellwagner and Reinefeld [6], October 1999. 118
- [8] 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. 115, 116
- [9] WWW: Smile Project Home Page. Shared memory in a LAN like Environment. <http://www.smile.in.tum.de/>, November 2001. 116
- [10] Y. Saad and M. H. Schultz. *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*. SIAM J.Sci. Stat. Comput., pp. 856–869, 1989. 115
- [11] C. Trinitis, M. Eberl, and W. Karl. Numerical Calculation of Electromagnetic Problems on an SCI Based PC-Cluster. In *Proceedings of International Conference on Parallel Computing in Electrical Engineering*, pages 166–170, Trois Rivières, Quebec, Canada, August 2000. International Conference on Parallel Computing in Electrical Engineering. 115, 119
- [12] Carsten Trinitis, Martin Schulz, Michael Eberl, and Wolfgang Karl. SCI-based LINUX PC-Clusters as a Platform for Electromagnetic Field Calculations. In *Proceedings of the Sixth International Conference on Parallel Computing Technologies (PaCT-2001)*, pages 510–513. Springer Verlag, 2001. 115

Application of a Parallel Virtual Machine for the Analysis of a Luminous Field

Leszek Kasprzyk, Ryszard Nawrowski, and Andrzej Tomczewski

Poznań University of Technology, Institute of Industrial Electrotechnology
60-965 Poznań, ul. Piotrowo 3A, Poland
{Leszek.Kasprzyk,Ryszard.Nawrowski,Andrzej.Tomczewski}@put.poznan.pl

Abstract. The article presents an exemplary solution of computation dissipation of luminous flux distribution in interiors, taking into account their multiple reflections, by means of a parallel virtual machine of cluster type. Communication between the processors was ensured by the PVM library operating under the operational system LINUX. Advantages and faults of the programs operating at the computers connected into the network making the cluster are discussed. An analysis of acceleration and efficiency of a concurrent program is carried out. During construction of the algorithm attention was paid to minimization of the number of transferred messages. Calculation examples are presented.

1 Introduction

The development rate of parallel and dissipated computation increases, in result of growing possibilities of fast computer networks for connecting considerable numbers of workstations of important calculation power [1]. Recently the need for solving sophisticated technological tasks increases, making use of modern numerical methods, which allow, in relatively short time, determining the results with high accuracy.

In many of them the parallel algorithms may be applied, having the following basic advantages:

- acceleration of the calculation process,
- improved reliability or accuracy of the operation,
- enabling to solve the tasks of a size exceeding the capacity of sequential computers.

The above is reflected in development of application of numerical methods, particularly in the field analysis methods. Denser discretization mesh considerably affects structure mapping of the analyzed systems and accuracy of obtained results [2]. However, such an approach results in increasing computation time, according to the complexity of the analyzed object [3].

Software designers aim at meeting the needs, making available a wide gamut of products enabling connecting computers into a network without comprehensive

knowledge of communication processes. This enables creating a virtual computer, i.e. a cluster. Availability of libraries for interprocessor communication induced growing popularity of programming tools serving concurrent and dissipated computation processes.

2 Characteristics of a Parallel Computer

In the Institute of Industrial Electrotechnology of the Poznań University of Technology a parallel virtual computer was built. At present its structure includes 9 computers configured as follows: a processor Intel Pentium III 866 MHz, RAM 256 MB and main plate Intel D815EPEA2. Connection of the nodes is ensured by a network created in the Fast Ethernet technology 100Mbps, with a 3COM switch. The above described homogeneous parallel virtual machine has operational power comparable to the computer IBM SP2 (4,2 GFLOPS) composed of 1 wide node (266 MHz, 128 MB RAM) and 14 narrow ones (266 MHz, 64 MB RAM).

All the metacomputer nodes work under the operational system Linux SuSE 7.2. The choice of the system was imposed by its stability, efficiency, and availability of the software necessary for connecting the computers into the cluster and implementation of parallel processing. For purposes of program realization a PVM software (Parallel Virtual Machine) was used. The environment allows execution of the programs dissipated at multiprocessor machines equipped with local, common memories, and at heterogeneous networks of sequential and parallel machines [4]. Additionally, one of the computers, playing a supervising role, has an XPVM software, enabling detailed analysis of operation of the parallel virtual machine in the graphic system X Windows.

3 Paralleling of Calculations of the Luminous Flux Distribution in Interiors

For purposes of the presented calculation method of luminous flux distribution in interiors the boundary integral method was used. The whole boundary surface of analyzed interior was divided into N elementary surfaces dS [3].

Every elementary surface dS with a reflection coefficient $\rho > 0$, reflects a part of the luminous flux Φ' falling to it directly from the source. In result of multiple reflections of the luminous flux between the parts of the surface S the total luminous flux Φ falling to the surface is equal to the sum of direct Φ' and indirect Φ'' components:

$$\Phi = \Phi' + \Phi'' \quad (1)$$

Taking into account the definition of utilization (coupling) factor f_{ji} of the i -th and j -th elements of the S surface one is able to determine the value of the indirect luminous flux value falling to the i -th element and coming from the j -th element [3].

Considering the effect of all elementary surfaces and the relationship (1) enables determining the value of total luminous flux falling to the i -th element of the

surface S . For the factor $i=1,2,\dots,N$ a system of linear equations may be formulated with unknown total luminous fluxes corresponding to the elementary surfaces dS_i . Ordering of the expressions gives:

$$\begin{bmatrix} 1 & -\rho_2 f_{21} & \dots & -\rho_N f_{N1} \\ -\rho_1 f_{12} & 1 & \dots & -\rho_N f_{N2} \\ \dots & \dots & \dots & \dots \\ -\rho_1 f_{1N} & -\rho_2 f_{2N} & \dots & 1 \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \dots \\ \Phi_N \end{bmatrix} = \begin{bmatrix} \Phi'_1 \\ \Phi'_2 \\ \dots \\ \Phi'_N \end{bmatrix} \quad (2)$$

where: Φ'_i - direct luminous flux falling to the i -th element, Φ_i - total luminous flux falling to the i -th element, ρ_i - coefficient of reflection of the i -th surface element, f_{ji} - utilization factor of the i -th and j -th elements dS .

Due to ill-conditioning [5] of the system of equations (2) an additional equation must be introduced. The assumptions already made in [3] yield that the sum of direct fluxes Φ'_i at all elementary surfaces dS_i is equal to the sum of luminous fluxes Φ_{srp} of all l sources included in the considered interior:

$$\sum_{i=1}^N \Phi'_i = \sum_{p=1}^l \Phi_{srp} \quad (3)$$

Insertion of the equation (3) into the system of equations (2) gives a system of equations possessing an explicit solution [5]:

$$\begin{bmatrix} 1+\rho_1 f_{1N} & -\rho_2 f_{21}+\rho_2 f_{2N} & \dots & -\rho_N f_{N1}-1 \\ -\rho_1 f_{12}+\rho_1 f_{1N} & 1+\rho_2 f_{2N} & \dots & -\rho_N f_{N2}-1 \\ \dots & \dots & \dots & \dots \\ 1-\rho_1 & 1-\rho_2 & \dots & 1-\rho_N \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \dots \\ \Phi_N \end{bmatrix} = \begin{bmatrix} \Phi'_1-\Phi'_N \\ \Phi'_2-\Phi'_N \\ \dots \\ \Phi'_{zr} \end{bmatrix} \quad (4)$$

As the illuminated interiors may have concave elements (of inner surface S_j and coefficient of reflection ρ) the calculation algorithm must be modified. For such surfaces the total flux that is reflected from and leaves the elementary surface is smaller than for a planar element. This is due to an internal coupling: a part of the flux is reflected several times within the same concave surface and attenuated. One of the methods enabling consideration of the fact in the system of equations (4) consists in reduction of the flux reflected from the concave element in the degree equal to the ratio of the closing surface S_{jo} (the surface of a plane closing the concave element) to total surface of the element S_j . In such a case the system (4) should allow for a relationship between the flux leaving the interior (Φ'_{ji}) and the flux falling to the surface of the interior (Φ_j) in the form:

$$\Phi'_{ji} = \rho_j \Phi_j \frac{S_{jo}}{S_j} \quad (5)$$

where: S_{jo} - closing surface area (area of planar surface closing the j -th concave element), S_j - total area of the j -th concave element.

Once the relationship (5) is introduced to the system (4), the final form of the system of equations describing the luminous flux in interiors, taking into account multiple reflections, is as follows:

$$\begin{bmatrix} 1+\rho_1 \frac{S_{i0}}{S_1} f_{1N} & -\frac{S_{20}}{S_2} (\rho_2 f_{21} - \rho_2 f_{2N}) & \dots & -\rho_N \frac{S_{N0}}{S_N} f_{N1} - 1 \\ \frac{S_{i0}}{S_1} (\rho_1 f_{12} - \rho_1 f_{1N}) & 1+\rho_2 \frac{S_{20}}{S_2} f_{2N} & \dots & -\rho_N \frac{S_{N0}}{S_N} f_{N2} - 1 \\ \dots & \dots & \dots & \dots \\ 1-\rho_1 \frac{S_{i0}}{S_1} & 1-\rho_2 \frac{S_{20}}{S_2} & \dots & 1-\rho_N \frac{S_{N0}}{S_N} \end{bmatrix} \begin{bmatrix} \Phi_1 \\ \Phi_2 \\ \dots \\ \Phi_N \end{bmatrix} = \begin{bmatrix} \Phi'_1 - \Phi'_N \\ \Phi'_2 - \Phi'_N \\ \dots \\ \Phi'_N \end{bmatrix} \quad (6)$$

where S_{i0} - closing surface area of the i -th elementary surface, S_i - total area of the i -th elementary surface.

Direct components of the luminous flux included in the system (6) are considered as a vector of free terms of the system of linear equations. Their values are determined numerically on the basis of the relations presented in the work [6].

In order to find a vector of total fluxes at the elementary surfaces dS_i the utilization factors occurring in the system of equations (6) should be determined. The i -th and j -th surface elements are located at the distance r each other. This allows to write that the utilization factor of the luminous flux f_{ji} is given by the expression [3]:

$$f_{ji} = \frac{1}{\pi S_j} \int_{S_j} \int_{S_i} \frac{\cos \alpha_j \cos \gamma_i}{r^2} dS_i dS_j \quad (7)$$

where: S_j - area of the j -th element, r - distance between the centres of the i -th and j -th elements, α_j - the angle between normal vector of the j -th surface element and a segment of the line connecting the centres of the i -th and j -th elements, γ_i - the angle between normal vector of the i -th surface element and a segment of the line connecting the centres of the j -th and i -th elements,

Detailed analysis of the numerical model has shown that at the first stage of the work two processes of the algorithm may operate parallel:

- calculation of a direct component of the luminous flux Φ' coming from all excitations (light sources) located in the interior,
- calculation of the utilization factors f_{ji} .

In the first process the principle of superposition is used (for the light field: the total luminous flux falling to the elementary surface dS is equal to the sum of direct luminous fluxes coming from all the sources located in the interior). Therefore, for l sources in the analyzed object the following expression may be written:

$$\Phi'_{dS} = \sum_{k=1}^l \Phi'_k \quad (8)$$

where: Φ'_{dS} - total direct flux at the element dS coming from all l sources, Φ'_k - the direct flux at the element dS coming from the k -th source.

In this case the calculation processes may be carried out simultaneously, dissipated among single excitations (light sources) or their groups.

Another process of the concurrent algorithm includes the calculation of the utilization factors (8). As an important number of surface integrals must be calculated (in the case of a dense calculation mesh) dissipation of the calculation process among bigger number of processors results in time saving.

For computer implementation of parallel calculation process a PVM software was used. It controls all the machines of the cluster and manages distribution of the processes among particular computers.

For purposes of numerical programs developed in C-language for the computers operating under Linux operational system a group of functions from the PVM library was applied, that enabled distributing of the tasks among particular nodes of the network and coordinating the messages including results of the calculations.

4 Examples of Calculations

In order to carry out the exemplary calculations an interior of industrial production bay of the dimensions 100×100 metres and the height 20 metres was selected, with symmetrically arranged 100 lighting fittings OPH 250. Every fitting included a light source (excitation) of the flux 20000 lm. Intensity distribution surface of the fitting was determined on the ground of measurements made by means of a computer arm-photometer. Additionally, the interior included minor architectural elements of well determined reflection parameters, screening the luminous flux.

Making use of the numerical programs in C-language with the concurrent elements, a series of calculations was made for a single cluster node and for several nodes. For the assumed number of elementary surfaces (90,000) the total computation time of distribution of luminous flux in the interiors was investigated. The selected number of discrete elements ensures the accuracy of obtained results corresponding to the type of the task being solved.

The relationship between the total computation time and the number of applied network nodes is shown in Fig. 1. In order to determine the effectiveness of the algorithm of concurrent calculations a basic parameter characterizing the parallel algorithms was used, i.e. the acceleration factor of the task. Its value is determined from the relationship:

$$S(n, p) = \frac{T(n, 1)}{T(n, p)} \quad (9)$$

where $T(n, p)$ is the time of execution for the same algorithm applied to the problem of the size n at the parallel machine composed of p processors, n – the size of the problem, p – the number of processors.

Moreover, the value of efficiency was defined as the ratio of the acceleration time for the problem of the size n performed on p processors to the number p of the processors.

The effect of the number of processors of the parallel virtual machine on acceleration factor (Fig. 2) and efficiency (Fig. 3) was determined and plotted, taking into account the equation (9) and the above definition.

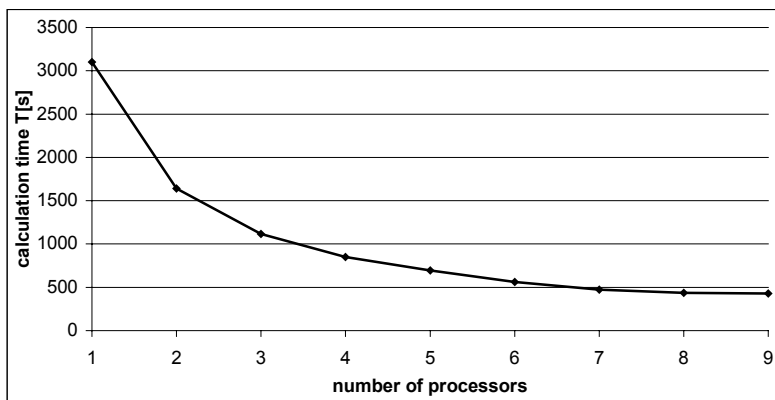


Fig. 1. Total calculation time $T[s]$ as a function of the number of processors

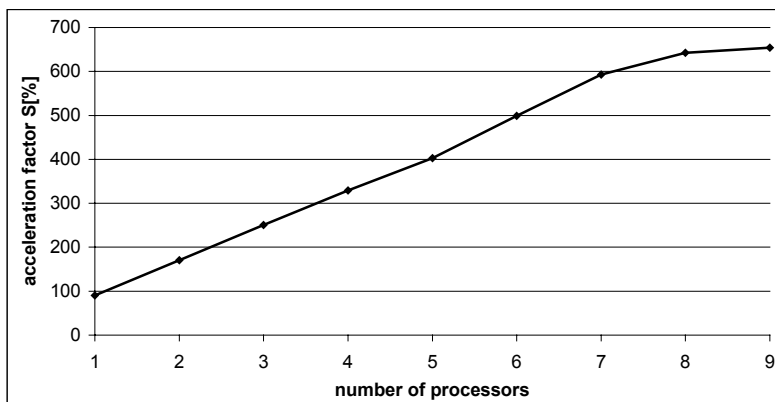


Fig. 2. Acceleration factor $S[\%]$ as a function of the number of processors

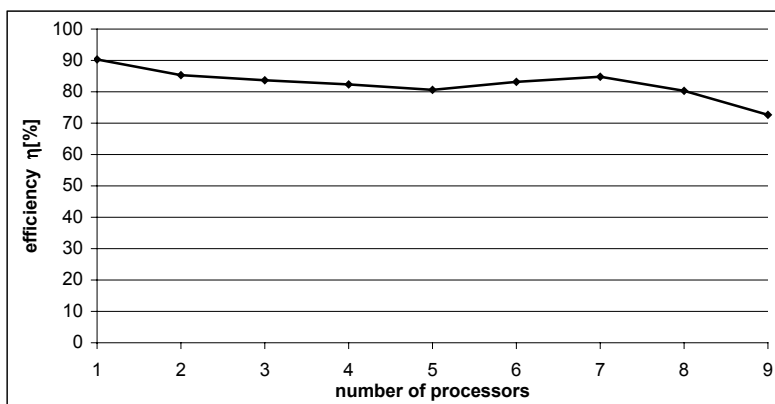


Fig. 3. Efficiency $\eta[\%]$ as a function of the number of processors

5 Final Notes and Conclusions

The calculations show that the use of a multiprocessor virtual machine for purposes of the analysis of luminous flux distribution in interiors is an effective method of reduction of total time of the calculation process. For fixed discretization parameters the concurrent calculation time of the luminous flux emitted by 100 sources decreased from 3100 s for a single processor (2802 s for a sequential one) to 408 s for 9 ones.

The considered computation task generates relatively small number of messages between the supervising computer and the nodes processing particular operations. This enabled stating that the value of the acceleration factor of the system (so-called efficiency) approximates an ideal value (Fig. 2). The algorithm appropriately dissipated the calculation process, with optimal application of all the processors.

Almost linear characteristics of the relationship between the acceleration factor and the number of processors (Fig. 1) is a result of the use of equal node hardware and evidences appropriate paralleling of the algorithm.

Efficiency of the calculations, shown in Fig. 3, approximates 90 per cent, mainly due to small number of messages transferred during the processing. Moreover, data transfer time is minimal as compared to the calculation time. This means that further research should be devoted to the analysis of the calculations carried out at parallel machines composed of a dissipated computer network and with the use of a heterogeneous cluster containing several smaller computers (taking into account their calculation power. Oscillation of efficiency with respect to the number of processors is worthy of notice. The effect is particularly visible while analyzing luminous flux distribution at the cluster composed of 7 machines. It is a results of a lack of full balance between the loads of particular processors, due to indivisibility of the tasks. Therefore, at the end stage of the calculation process some processes are inactive. In result it was assumed that further algorithms should be examined taking into account the loads of particular computation nodes, minimizing the number of inactive computers. It is of particular significance in the case of heterogeneous networks.

It was stated that calculation performance of metacomputers including the machines connected with computer cluster with low transfer capacity lines in a widespread network are worthy of research. Dissipation of the calculations made this way, although apparently ineffective, is of considerable efficiency in the case of calculation processes in which the principle of superposition may be used. In practice, such a task may be dissipated from the very beginning, while, having finished the calculations, the results may be transferred to a supervising process. In result the communication does not significantly affect the total calculation time.

References

- [1] Jordan, A.: Theoretical Background of Parallel Electromagnetic Field Computations, The 10th International Symposium on Applied Electromagnetics and Mechanics, Tokyo (2000) 193-194
- [2] Bertsekas, D. P., Tsitsiklis, J. N.: Parallel and Distributed Computation: Numerical Methods, Prentice Hall (1989)

- [3] Nawrowski, R.: The use of multiple reflection method for calculation of luminous flux in interiors, *Journal of Light & Visual Environment*, Vol. 24, No 2, (2000) 44-48
- [4] Geist, G. A., Kohl, J. A., Papadopoulos, P. M.: PVM and MPI: a Comparison of Features, *Calculateuris Paralleles*, Vol. 8, No 2, (1996)
- [5] Burden, R. L., Faires, J. D.: *Numerical Analysis*, Weber & Schmidt, Boston (1985)
- [6] Gwynne, E.: *Practical Numerical Analysis*, John Wiley & Son, New York (1995)

Solving Engineering Applications with LAMGAC over MPI-2^{*}

Elsa M. Macías and Alvaro Suárez

Grupo de Arquitectura y Concurrencia (G.A.C.)
Departamento de Ingeniería Telemática
Las Palmas de Gran Canaria University (U.L.P.G.C.), Spain
{elsa}@cic.teleco.ulpgc.es
{alvaro}@dit.ulpgc.es

Abstract. Traditionally, the Local Area Network (LAN) has been used for parallel programming. However, with recent advances in wireless communications, some authors think that the future of Wireless Local Area Networks (WLAN) depends on the demonstration of its application to real problems. In this paper we prove that wireless communications are suitable to perform parallel computing by presenting interesting experimental results for two engineering applications that follows the Master-Slave style programming with data dependencies among iterations. We consider a dynamic variation of slave processes as new portable nodes are added to the WLAN or detached from it.

1 Introduction

There is no doubt that the use of wireless technology is increasing by leaps and bounds. Last advances have provided powerful portable computers with radio network interface cards that operate at 2, 11, 25 and shortly 54 Mbps [1]. Despite of the technical improvements, the traditional networking will not be replaced by the wireless one but the promising benefits are in the combination of both of them. For example scalable parallel applications can reduce the execution time if the number of computing resources increases. This situation may be viable in a wireless network-based computing environment such as a LAN-WLAN infrastructure in which there are dedicated computers interconnected with wires and a potential number of portable computers available to perform computations when the owners add them to the computing infrastructure.

The use of MPI or PVM is feasible in wireless communications because they are built on top of TCP/IP and therefore the physical medium does not impose any restriction. However, the technical challenges in designing parallel applications for these environments are quite different from those involved in the design of applications for traditional network based computing. For example, a great concern in wireless communication is network failure and high error rates. In the

^{*} Research partially supported by Spanish CICYT under Contract: TIC01-0956-C04-03.

literature there are some approaches that deal with network, node or process failures [2][3][4][5] but some of them introduce high overheads in the network, others consider applications with trivial parallelism or else the adopted solution is not appropriate for a LAN-WLAN infrastructure. Another challenge is the management of the parallel virtual machine from an MPI application as new nodes are added or detached.

There are some related works that combine traditional LAN with WLAN to perform computing. For example, in [6] the authors consider wireless and mobile devices as significant computational resources for the Grid and present the Legion infrastructure to support these devices. They state that Legion supports MPI but they do not mention how MPI applications under Legion system can manage dynamic changes in the parallel virtual machine or wireless channel disconnections. In [7] the authors present a hybrid cluster with stationary and mobile computers showing a deterioration of the parallel program performance because of the great communication overhead, especially over the wireless links. We have defined a LAN-WLAN architecture and a programming model for the execution of parallel applications on that infrastructure [8]. In [9] we presented a library named LAMGAC that helps the programmer in the development of MPI parallel applications for a LAN-WLAN infrastructure and the WEB interface to access to the system (job submission and attachment/detachment of computers). We manage the dynamic variation of the parallel virtual machine during the execution of MPI parallel programs and at present we also detect wireless channel disconnections to let the parallel program continue gracefully to its ending. In this paper, we present interesting experimental results for two Master-Slave engineering applications that make evident an alternative use of wireless networks.

The rest of the paper is organized as follows. In section 2 we briefly review the architecture of our system. Section 3 is devoted to present the engineering applications. Then it is presented some novel experimental results. Finally we sum up the conclusions and present directions for further research.

2 Brief Revision of the System Architecture

The overall structure of the computing architecture is presented in Fig. 1. We consider Master-Slave parallel applications in which there is a main loop and for every iteration the processes must synchronize to interchange data. The master is in the access node (*AN*) and the slaves are in the Wired Nodes (*WN*) and Portable Nodes (*PN*).

The variation of the parallel virtual machine may be checked before starting next iteration (PN are detached from (or added to) the WLAN due to user's intervention or physical disconnection, not implementing a cycle-stealing mechanism [10] because it could flood the wireless channel with beacons and reduce the available bandwidth). We use a WLAN with infrastructure, so PN only communicate with AN (following the Master-Slave model). For this reason, the master executes the attachment and detachment protocol and only this process

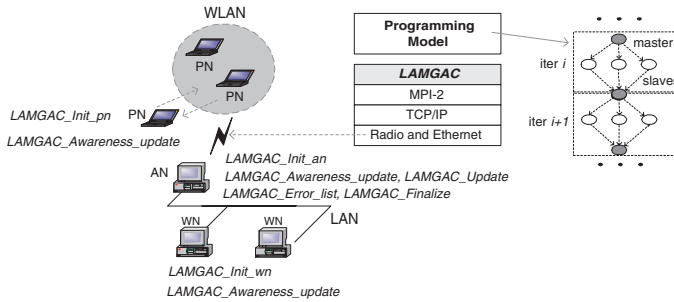


Fig. 1. Hardware and software architecture of our computing system

can spawn slave processes creating a new intercommunicator for each spawning. This is the most efficient mechanism to tolerate total or sporadic network disconnections. In Fig. 1 the names of LAMGAC functions that support this programming model are shown. The slave processes have or not awareness about the overall number of processes that makes computations depending if they invoke *LAMGAC_Awareness_Update* or not.

3 Description of the Engineering Applications

A general scheme for both applications is presented in Fig. 2. The master process is in charge of recollecting data from slave processes, then it may make intermediate computations, and finally it broadcasts data for doing computations in next iteration. To update the number of processes that collaborates per iteration it is invoked *LAMGAC_Awareness_update* because of the data distribution for next iteration may be done with the returned information by this function (process' rank and number of processes for next iteration) and the broadcasted data by master process.

3.1 Unconstrained Global Optimization for n-Dimensional Functions

The problem consists of finding the global minima for a n-dimensional function using a strategy based on a branch and bound methodology that recursively splits the search domain into smaller and smaller parts that we name *boxes* (Fig. 3a). For each iteration, master process communicates the search domain to slave processes. Then, the processes select a set of boxes and explore them searching for minima values. These values are gathered by master process which selects the box containing the smallest minimum so far and the boxes which contain a value next to the smallest minimum. All the other boxes are deleted. The loop is repeated until the stopping criteria is satisfied. The volume of communication per iteration (Eq. 1) varies proportionally with the number of processes and search domains (the number of domains to explore per iteration is denoted as

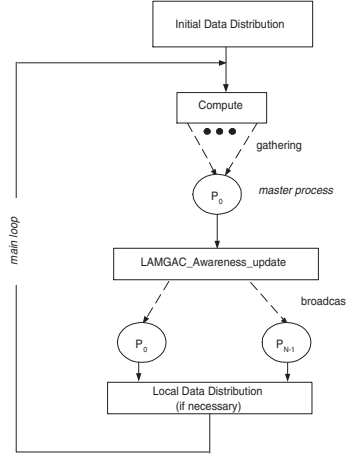


Fig. 2. General scheme of the applications

$dom(i)$ which is decomposed in boxes locally by a process knowing its rank and the boundaries of the subdomain).

$$communication(i) = A_1 * dom(i) * [1 + PN(i)] + A_2 * \sum_{k=1}^{PN(i)+WN} min(i)^{P_k} + B \quad (1)$$

A_1 represents the cost to send the boundaries (float values) for each subdomain (broadcast operation to processes in the LAN and point to point sends to slave processes in PN), $PN(i)$ is the number of processes in the WLAN in the iteration i , $min(i)^{P_k}$ is the number of minima (integer value) calculated by process with rank k in the iteration i , A_2 is the data bulk to send the computed minimum to master process (value, coordinates and box, all of them floats), and B is the communication cost for LAMGAC_Awareness_update (sending process' rank and number of processes from master to each slave process).

3.2 Simulation of an IR Diffuse Communication Channel

The simulation consists of computing the temporal power distribution that reach the optical receiver for a short time due to the emission of a light impulse generated in an optical source. Both, the optical receiver and source are located inside a room operating the walls of the room as reflector surfaces (we refer them as *cells*). The power reaching the receiver is direct (if there is a non-interruption link of sight path between the source and the receiver) or indirect because of the different rebounds on the cells before reaching the receiver. In the former, the power is higher and the delay is shorter. The distribution technique consists of distributing the cells of a wall among the processes, repeating this technique

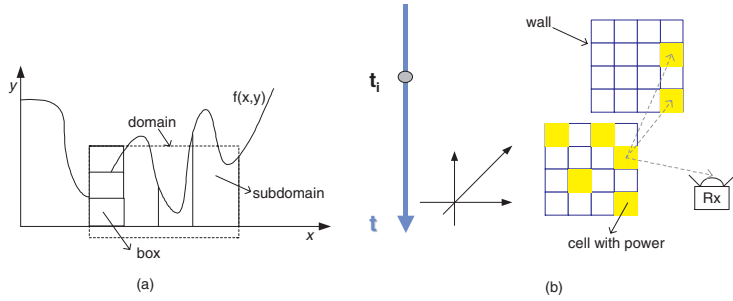


Fig. 3. Data distribution: a) optimization problem b) IR diffuse channel

for each wall of the room. This initial data distribution only changes whether LAMGAC_Awareness_update communicates a variation of the parallel virtual machine. For every iteration (time t_i in Fig. 3.b), the processes explore a set of cells selecting those with power. The power reflected from these cells to other cells or even to the receivers is computed. Then, the power distribution in the walls for the next discrete time of simulation (next iteration of the main loop) is gathered by master process. This process makes the sum of the power values computed by individual processes and broadcast these power values to all processes. The data bulk communicated per iteration is equal to

$$communication(i) = 2 * cells * [1 + PN(i)] + B \quad (2)$$

that represents the cost to send twice the power (float value) in the cells of the room (size equal to *cells*): one sending is from the slave processes to the master and the other one is from the master to the slave processes.

4 Experimental Results

The characteristics of machines are: AN, 300 MHz dual Pentium II; WN1, SMP with four 450 MHz Pentium II; WN2 and WN3, 667 MHz Pentium III; WN4, 200 MHz Pentium MMX; PN1, 667 MHz Pentium III; PN2, 550 MHz Pentium III. All the machines run under LINUX operating system. The wireless cards complies with IEEE 802.11 standard at 2 Mbps while LAN speed is 10 Mbps. The input data for the optimization problem are the following: *Shekel* function for 10 variables [11], initial domain equal to $[-50, 50]$ for all the variables and 100 random points per box. For the IR channel it was simulated the configuration A presented in [12] that consists of a rectangular room with the source located at ceiling and pointed towards the floor and the receiver located at the floor and pointed towards the ceiling so that there is a direct line of sight path between them. The emission power is 1 watt and the simulation time is 60 nanoseconds. For all experiments we assume a null user load and the network load is the application's one. The experiments were repeated 10 times obtaining a low standard deviation.

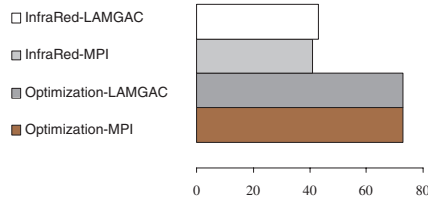


Fig. 4. MPI program versus LAMGAC program (VM_LAM0 configuration)

The sequential time for the optimization problem is 167'13" and 114' for the IR channel simulation.

4.1 LAMGAC Library Overhead

The execution times of the MPI parallel program and the equivalent LAMGAC parallel program were measured (in the latter without attachments and detachments of processes to make comparisons). In Fig. 4 it is presented the timings in minutes for both engineering applications when using the virtual machine configuration called VM_LAM0 that consists of AN and the nodes in the LAN. LAMGAC_Awareness_update is called in the optimization problem 7 times versus 170 in the infrared channel simulation. That is the reason why the time difference is higher between the MPI program and the LAMGAC program for the IR channel problem. As you can see, experimental results show a negligible overhead when using LAMGAC library.

4.2 Computing with Wireless Nodes

Experimental results show that the execution time of LAMGAC algorithms is rather similar if the number of processes is the same for different simulations using wired communications or mixing wired and wireless communications. The results in minutes are presented in Fig. 5. Notice that the execution time of the LAMGAC optimization program for VM_LAM1 or VM_LAM2 configuration is minor than the result presented for configuration VM_LAM0 despite the fact that the number of processors for the VM_LAM0 configuration is higher. This is because the lower power processor (WN4) increases the overall execution time (take into account the high amount of computations and the process synchronization in the optimization problem). The same comments are applied for the IR channel simulation.

The execution time decreases if a process is spawned in a new PN. On the other hand, if the attached process is detached from the parallel program, the

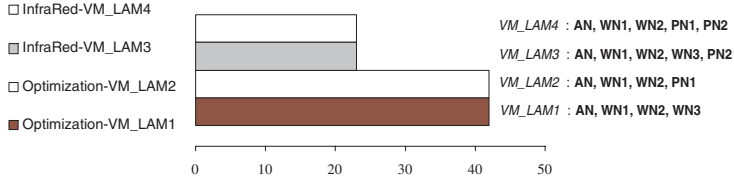


Fig. 5. LAMGAC execution times for different configurations

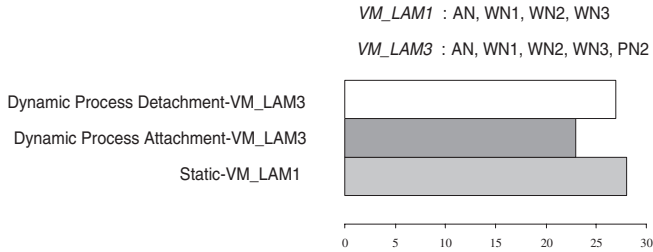


Fig. 6. Infrared channel simulation with attachment and detachment

overall execution time of the program increases. Both results in minutes are presented in Fig. 6 for the IR channel simulation. The attachment was made in the second iteration of the parallel program. The detachment took place about 10 minutes after the attachment. The results are similar for the optimization problem: the execution time for VM_LAM2 configuration when a process is detached in the fourth iteration is 46'42" versus 42'14" without detachment.

5 Conclusions and Future Research

In this paper we have proved that the WLAN may be a useful infrastructure for doing parallel computing for Master-Slave parallel applications. Experimental results show that after adding a portable node to the parallel virtual machine and spawning a process to collaborate with the current computations of the parallel program, the overall execution time is reduced. On the contrary, if a previously attached process is detached from the parallel program, then the execution time increases.

At present, we have implemented a simple mechanism to detect disconnections of the wireless channel, which is a great concern in wireless communications and it may lead the whole MPI application to abort. In our ongoing work, we

are reprogramming the engineering applications to provide them with the fault detection mechanism and testing the performance of this new LAMGAC functionality. On the other hand, we are also planning to add dynamic load balancing schemes in our library considering the heterogeneous characteristics of the computing environment: processor power, communication speed and external load.

References

- [1] IEEE Standards Wireless Zone: <http://standards.ieee.org/wireless/> 130
- [2] Agbaria, A. M., Friedman, R.: Startfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. 8th IEEE International Symposium on High Performance Distributed Computing (1999) 131
- [3] Fagg, G. E., Dongarra, J. J.: FT-MPI: Fault Tolerant MPI Supporting Dynamic Applications in a Dynamic World. Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users Group Meeting. Balatonfüred, Hungary. LNCS 1908. Springer Verlag (2000) 346-353. 131
- [4] Louca, S., Neophytou, N., Lachanas, A., Evripidou, P.: MPI-FT: Portable Fault Tolerance Scheme for MPI. Parallel Processing Letters, Vol. 10, N^o4 (2000) 371-382. 131
- [5] Stellner, G.: CoCheck: Checkpointing and Process Migration for MPI. IEEE International Parallel Processing Symposium (1996) 526-531. 131
- [6] Clarke, B., Humphrey, M.: Beyond the "Device as Portal": Meeting the requirements of Wireless and Mobile Devices in the Legion Grid Computing System. 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing, 15th April, Fort Lauderdale, Florida (2002) 131
- [7] Cheng, L., Wanchoo, A., Marsic, I.: Hybrid Cluster Computing with Mobile Objects. 4th IEEE International Conference/Exhibition on High Performance Computing in Asia-Pacific. Beijin (China) (2000) 909-914. 131
- [8] Macías, E. M., Suárez, A., Ojeda, C. N., Gómez, L.: A Novel Programming Strategy Considering Portable Wireless Nodes in a Heterogeneous Cluster. Distributed and Parallel Systems. From Instruction Parallelism to Cluster Computing. Kluwer Academic Publishers (2000) 185-194. 131
- [9] Macías, E. M., Suárez, A., Ojeda, C. N., Robayna, E.: Programming Parallel Applications with LAMGAC in a LAN-WLAN Environment. Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users Group Meeting. Santorini, Greece. LNCS 2131. Springer Verlag (2001) 158-165. 131
- [10] Heymann, E., Senar, M. A., Luque, E., Livny, M.: Evaluation of an Adaptive Scheduling Strategy for Master-Worker Applications on Clusters of Workstations. High Performance Computing, 7th International Conference, Bangalore, India, LNCS 1970. Springer Verlag (2000) 310-319. 131
- [11] Java-based Animation of Probabilistic Search Algorithms: http://goethe.ira.uka.de/people/syrjakow/websim99_w434/websim99paper.html 134
- [12] Barry, J. R., Kahn, J. M., Krause, W. J., Lee, E. A., Messerschmitt, D. G.: Simulation of Multipath Impulse Response for Indoor Wireless Optical Channels. Proceedings of the IEEE Journal on Selected Areas in Communications. Vol. 11 (3) (1993) 367-379. 134

Distributed Image Segmentation System by a Multi-agents Approach (Under PVM Environment)

Yacine Kabir ¹ and A. Belhadj-Aissa ²

¹ Signal and image processing Laboratory (LTSI)
Welding and testing research centre
Route de Dely Ibrahim, B.P.64, Chéraga, Algiers, Algeria
Tel./Fax: (213) 21 36 18 50
kyac@wissal.dz

² Image processing and radiation Laboratory
Electronic and computer science Department, USTHB University, Algeria
h.belhadj@mailcity.com

Abstract. We propose to conceive a distributed image analysis system, according to a multi-agents approach. We are especially interested in the first steps system, (image pre-processing and segmentation). We introduce a new segmentation system obtained by the co-operation of a robust edge detector of canny type and a region growing. The cooperation technique allows exploiting the advantages of each method to reduce the drawback effects of the other one. A multi-agents approach provides an increased flexibility, by separating the various expertises within the system. Each agent is responsible for a precise task and its interactions with other agents are carried out by well defined protocols. Such an implementation makes the system development easier as well as later modification. The developed architecture allows carrying out in parallel a lot of system processes. The system consists of several agents executed as independent processes on several machines connected on a heterogeneous network and communicating through message passing using PVM.

Keywords: computer vision, image segmentation, distributed processing, multi-agents, PVM.

1 Introduction

Image Analysis and interpretation algorithms in computer vision are sometimes very complex and involve an important computational load, on one hand, this is due to a huge information quantity handling, related to image data and processing parameters, and on the other hand one have to manage several knowledge sources. This complexity is more significant as the analysis, recognition and decision methods require more

and more complex and advanced mathematical techniques such as artificial intelligence techniques as well in low level processing or high level.

An image analysis system must be conceived according to a very flexible, robust and adapted architecture to allow reasonable processing times and to reduce the algorithms complexity. In this paper, we propose to conceive a distributed image analysis system dedicated to radiographic films according to a multi-agents approach.

The idea beyond this work consists in distributing the expertise within a society of entities called agents, both control operations and data are distributed. Each agent has its own competences, but it needs to interact with others in order to solve problems depending on its expertise field and to avoid conflictual situations.

In fact, the purpose of the system is to cooperate several algorithms sequentially or in parallel, at different processing stages. The system is distributed according to a network of heterogeneous machines.

An agent is defined as being an autonomous entity, able to act on it as well as on its environment. An agent has a partial representation of its environment and communication capacities with other agents, its behaviour is the consequence of its observations, its knowledge and its interactions with other agents. A multi-agents system allows solving problems by communication between several agents, without obvious external control, like an animal or human society in which individuals are brought to coordinate their activities and to negotiate to cohabit and to solve conflicts. Without this cooperation an individual could hardly or impossibly survive.

2 Approach Motivation

The use of a multi-agents approach in a parallel image analysis system have got some evident advantages [1][2][10].

Flexibility: A multi-agent approach provides an increased flexibility, by separating the various expertises within the system. In this framework, one can plan to have specialized agents to estimate image characteristics, to enhance image quality, to apply an edge detection operator or to apply region segmentation operator. A pre-processing agent specialised in image enhancement can be based on the information provided by an image estimation agent (noise, contrast, texture...).

Each agent is thus responsible for a precise task and its interactions with other agents are carried out by well-defined protocols. Such an implementation makes the system development easier as well as a later modification. For example, if one wants to modify the edge detection operator, it is enough to change the agent responsible for this processing only. The other agents should not necessarily be modified.

Parallelism: Parallelism can be considered at several processing stages, for example in the image characteristics estimation stage; indeed, several agents containing various types of estimators can work in parallel on the initial image and the results can be merged by another agent. In the same way parallelism can be considered in the segmentation process between edge detection and a region growing.

Cooperation: The cooperation can be considered between an edge detection agent and a region-growing agent in order to carry out a cooperative segmentation.

3 Image Segmentation

Image segmentation is a fundamental stage in any artificial vision system. It allows producing a compact description that can be characterized by edges or by homogeneous regions [5]. Image edges represent very rich information and have been extensively studied. The "region" approach is complementary to the "contour" approach, however it does not provide the same results. There is no unique segmentation, for each application, it's necessary to choose the appropriate technique, with the adapted parameters. Each technique has its advantages, drawbacks and limitations. The aim of each one is closely related to the application in question.

A comparison between the two approaches (edge and region segmentation) has shown that neither edge or region segmentation can provide alone an ideal segmentation. The future of image analysis is probably in the cooperation between these two approaches. A cooperation technique allows exploiting the advantages of each method in order to reduce the drawback effects of the other one. We will present in this paper a new method by combining estimation, enhancement, edge detection and a region growing in an intelligent process.

4 Proposed Method

The segmentation is obtained by the cooperation of a Canny edge detector [6] and a region growing. We have introduced an adaptation stage allowing a better choice of the edge detector parameters (scale, binarisation thresholds) and the homogeneity criteria thresholds related in a great part to the textured aspect of the image[4][8].

The segmentation method is performed according to the following stages:

Estimation stage

- Building the local contrast map; image points are classified into 3 categories: zone highly contrasted (ZHC), homogenous zone (HZ) and weakly contrasted zone (WCZ).
- Creation of a map characterising the homogeneity image surface, this allows the localisation of textured zones. Image points are classified into 2 categories: uniform zone (UZ) and non-uniform zone (NUZ):
- Noise estimation, we consider three types of noise, additive, multiplicative and impulsive.

Image pre-processing

- Contrast stretching if the image is poorly contrasted.
Image enhancement by filtering, according to the noise nature affecting the image (previously estimated):

- Construction of a preliminary edge map (Canny edge detector)[3][6], allowing to obtain high precision on image objects border;
The cooperation process begins by edge detection; the obtained edges are well localized, and correspond precisely to the real region borders. The segmentation quality depends largely on the region boundaries detected. The edge operator can generate false edges or miss edges due to a bad lighting.
- Small edge elimination (size varies with the application in question)
The edge detection operator is sensitive to all gray level variation, it can present wrong responses in textured or noisy zones; in fact, this operator doesn't take in consideration local characteristics; apart from this fact, sometimes, threshold values are not valid for the whole image. It's then very important to eliminate these false small edge chains, while respecting the type of image to segment, and the nature of mannered objects. At this step, there's a risk to eliminate true edge chains, and in spite of all, some small chains without significance are still not suppressed because of the threshold values used.
- Performing a region growing segmentation constrained by the edge map and extraction of the regions boundaries; the growing process start from seeds (randomly, extracted from a preliminary region segmentation...);
Growing region segmentation, detects better the content of objects in an image, because of the utilization of homogeneity criteria. These criteria allows detecting zones having the same texture attributes. It will be therefore possible to confirm the true contours due to a discontinuity between two distinct adjacent regions and to eliminate false edges, due to a sensitivity to micro-textures, to a degraded gray level, or to a shade. The growing process doesn't evolve beyond contours.

For the aggregation process, a double criteria is used; the growth is stopped when the pixel candidate to the aggregation:

- Doesn't verify the similarity criteria (homogeneity);
- Coincides with an edge pixel in the edge map.

Because of the edge constraint that the process of region growing undergoes, small regions having the measurements of small edge chains are be created

- Fusion of small regions, whose sizes, can be adjusted by the operator, or fixed according to an a priori content knowledge of the image treated. The region growing phase, generate a big number of small regions because of the choice of parameters used. The edge constraint in the last step can also give birth, to small isolated regions (figure 1). A region is identified as belonging to this false region type, if all its points coincide with edge points, therefore, We have two types of small regions to be merged:
 - Regions generated by edge constraint;
 - Regions generated by the criteria choice

If a small region must be suppressed, it's merged with the most similar adjacent region.

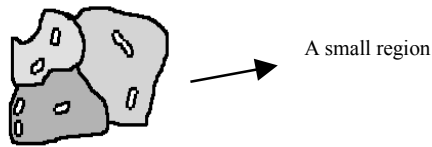


Fig. 1. Creation of small regions because of the edge constraint

- Verification of the common boundary between each pair of adjacent regions, in order to confirm their dissimilarity (fusion of over-segmented regions)

Over-segmented regions are generally owed to an insufficient illumination, to shades, or to degraded of color or light. Several rules based on simple and composed criteria are used to operate this verification. For example, two adjacent regions are merged if:

- The mean gradient along the common boundary is weak.
 - If there is not, sufficiently of edge points on the common boundary.
 - Elimination of small chains localized far from the boundary of any region.
 - Contour closing of the edge map controlled by the region segmentation map, broken edges are connected by prolongation on region boundaries (boundary following) (figure 2).
- Compatibility evaluation between the two maps, edge and region, using dissimilarity measures.

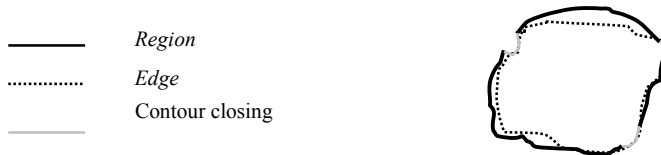


Fig. 2. Closing contour example by prolongation on region boundary

5 Multi-agents System Design

In this section, we propose a multi-agent implementation of the method described in the preceding paragraphs. It consists of several agents (16 in the current implementation) where each agent is an executable file. These agents are then executed as independent processes on several machines connected on a heterogeneous network (of different architectures and/or working under various operating systems).

The agents communicate between each other through message passing using PVM (figure 6), where PVM is an environment, which allows a heterogeneous network computers to be used as one virtual parallel machine. Several platforms are used to develop distributed systems, the choice of PVM platform has been adopted just because it's simple, easy to use and widely diffused [7]. The system has a vertical organization characterized by a structure having several levels, each level is struc-

tured in a horizontal way. The cooperation between agents can be summarized as follows [9]:

- To update the model of the agent surrounding world.
- To integrate information coming from other agents.
- To stop a plan in order to help other agents.
- To delegate the task which cannot be solved to another agent knowing its competences.

5.1 System Architecture

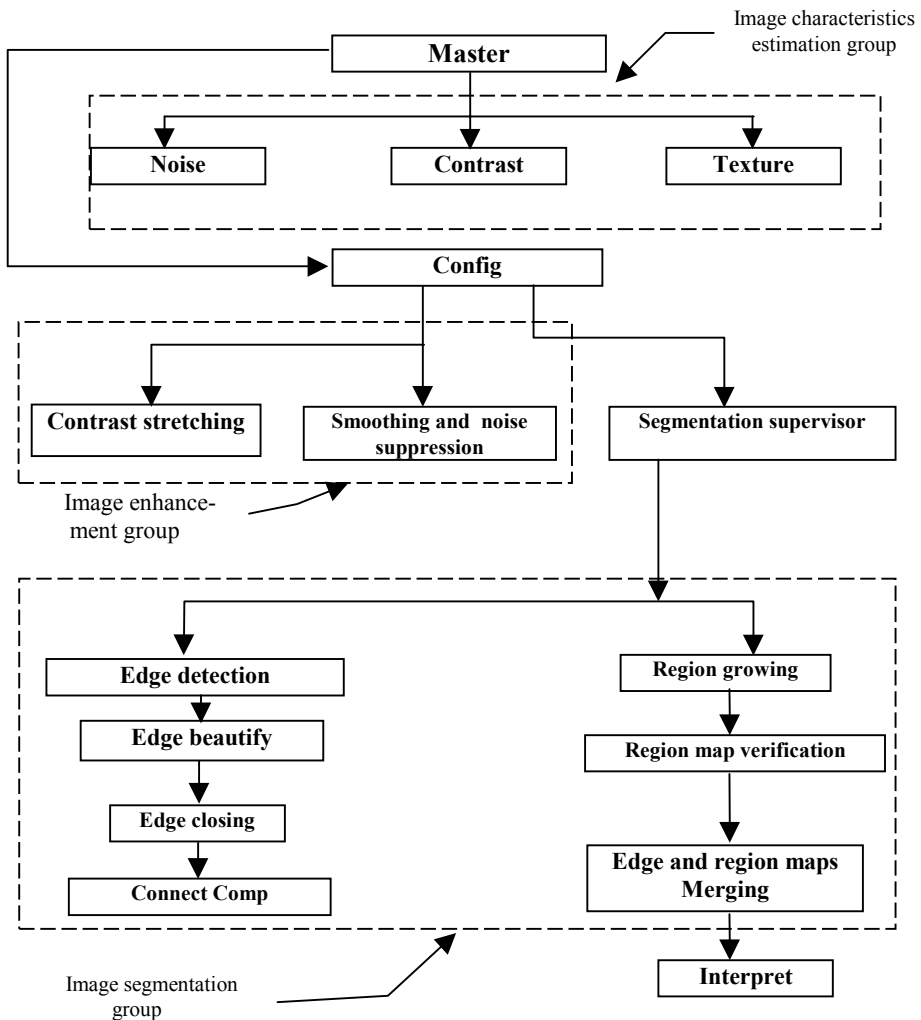


Fig. 3. Agent's creation synopsis

Figure 3 shows the different agents involved in the system. The system tasks are divided into several groups of agents having the following functionalities:

1. A user interface agent to read the initial image and the system parameters provided by the user (main agent).
2. A group of agents to calculate the adaptation parameters by estimating image characteristics (agent noise, agent contrast, agent textures). The work of these agents is carried out in parallel.
3. An agent to collect the estimation results in order to choose the adequate and necessary pre-processing (agent config).
4. A group of agents for pre-processing (accentuation agent, filter agent).
5. An agent to collect the pre-processing result and the segmentation parameters (agent seg).
6. A group of agents for image segmentation by the a cooperative method Edge-Region, this group is divided into two sub-groups of agents being carried out in parallel
7. Group of edge agents for an edge segmentation (Edge agent, beautify agent, edge closing agent, Connect Comp agent).
8. Group of region segmentation agents (Region agent, verif agent).
9. The processing is sometimes carried out sequentially in each type of group, the cooperation between the two groups is realized by the transmission of the edge map generated by the edge closing agent towards the verif agent and the transmission of the region map of the verif agent towards the edge closing agent.
10. A group of agents for collecting the cooperative segmentation results and interpreting the contents (fusion agent, interpret agent).

5.2 Communication between Agents

The communication between agents is selective (each one communicate with a precise number of others agents) and based on a message passing principle, the system is composed of social agents, each one need to evolve, this evolution can be seen as an interaction or a cooperation with other components in the system. An agent must need for at least some elementary communication functions, which enable it to carry out basic operations such as sending or receiving a message coming from another agent. The nature of a message depends on the type of the required task and handled information (image, filtering or edge detection parameters, estimation, a decision, testing a rule...); an agent needs sometimes to go through an intermediate agent to communicate with another agent.

6 Results and Discussion

A c++ library was written to model the proposed system, it's essentially composed of two parts, one for image processing routines, the second one for agents modelling and communication functions. The software development respect the generic programming

paradigm [10], this constraint allow having a reusable, flexible and machine independent code. The source code of agents can thus be compiled to run under a wide variety of operating systems and machines architectures.

We applied the developed segmentation method to several kinds of images; we present the segmentation results (figure 5) on two (2) images, The first image is an indoor scene, characterized especially by the presence of strongly polyhedric forms, having uniform grey surfaces, with presence of shades and badly enlightened parts (for example around the telephone wire); the second image shows a radiographic image of a welded joint presenting a defect (lack of penetration).

We can give some remarks relative to the segmentation quality, in spite of the contours prolongation on borders step; some gaps remain still not filled, because of the depth exploration choice to search candidate's points for the edge closing operation. All edge chains inside homogeneous regions are eliminated. The initial choice of the edge detection parameters is carried out to detect points with very strong gradients only; these points are useful to guide region growing process.

implemented method, showed its capacity to contribute in solving the segmentation problem through the various results presented. The adopted strategy is based on the mutual assistance of the two approaches edge and region, which made it possible to have a good localisation of objects contours, and to reduce the number of over-segmented regions, the criteria used wealth has also shown its efficiency. The results obtained indicate that the quality of detected contours influences the behaviour of the region growing process. It is necessary moreover to give a great importance to the pre-processing stage, which can reinforce the pixels resemblance inside the same area. An objective evaluation study (not mentioned in this paper) of the result was made, and showed that the error committed is acceptable. The result is perfectible, and other improvements can be made to make the technique more effective.

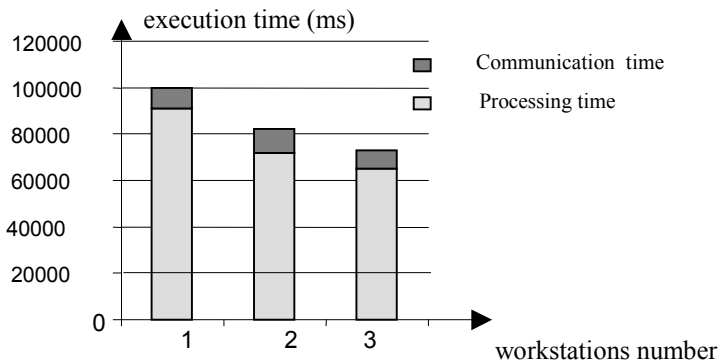


Fig. 4. Time evaluation (with office image)

Figure 4 indicates that the work load decrease with the number of workstations used in the virtual machine, it can be seen that the performance degradation is due primarily to the amount of inter-machines communication.

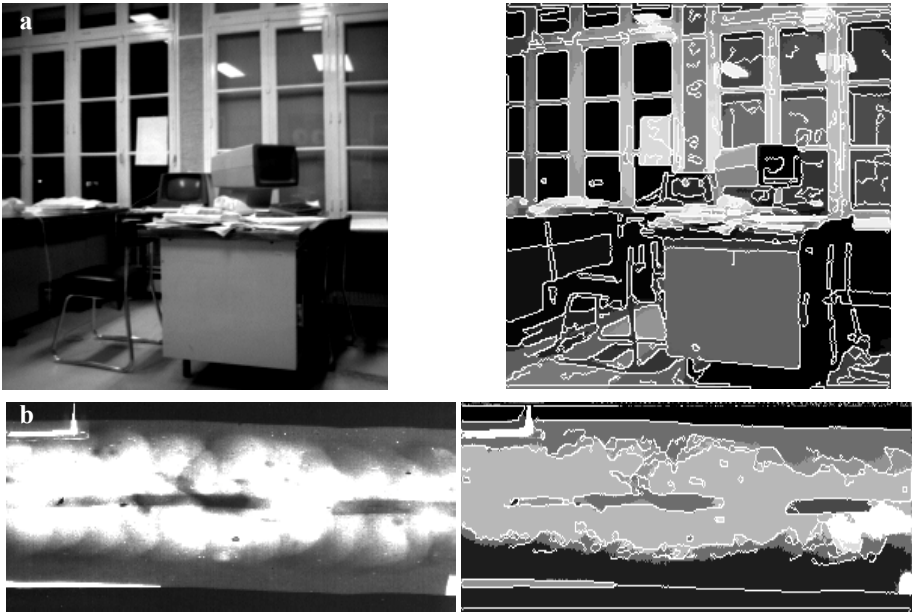


Fig. 5. Segmentation with the proposed method a) office image, b) welded joint image radiogram presenting a defect (Lack of penetration)

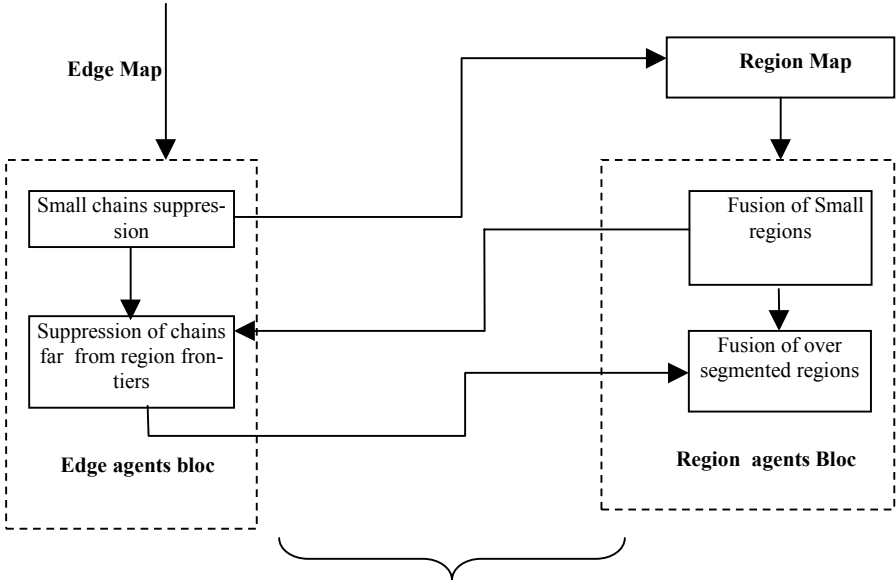


Fig. 6. Example of a mutual cooperation between region and edge agent blocks using message-passing negotiation

References

- [1] O. Baujard, Conception d'un Environnement de Développement pour la Résolution de Problèmes, Apport de l'Intelligence Artificielle Distribuée et Application à la Vision. PhD Thesis, Joseph Fourier university, GRENOBLE I, 1992.
- [2] A. Boucher, Une approche décentralisée et adaptative de la gestion d'information de vision, PhD Thesis, Joseph Fourier university, Grenoble 1. 1999.
- [3] J.F. Canny. Finding edges and lines in images. MIT Artificial Intel. Lab. Cambridge, MA, REP. AI-TR-720, 1983.
- [4] Y.Cherfa, Y.Kabir, R.Drai. X-Rays Image Segmentation for NDT of Welding defects. 7th ECNDT'98, May 26-29, 1998, Copenhagen, DENMARK.
- [5] J.P Cocquerez et S.Philipp. *Analyse d'images: Filtrage et Segmentation*, Edition Masson.
- [6] R. Deriche, Using Canny's criteria to derive a recursive implemented optimal edge detector, *International journal of computer vision*, pp. 167-187, 1987.
- [7] Al Geist et al. PVM: " A Users' Guide and Tutorial for Networked Parallel Computing", The MIT Press, Cambridge, Massachusetts, London, England, 1994.
- [8] Y.Kabir. X-rays image segmentation dedicated to non destructive testing. Master Thesis. Univ Blida, Algeria. 1999.
- [9] S.labidi & W.Lejouad, De l'intelligence artificielle distribuée aux systèmes multi-agents. Research Report n° 2004. INRIA.
- [10] M. Lückenhaus, W. Eckstein, "A Multi-Agent based System for Parallel Image Processing", in: *Parallel and Distributed Methods for Image Processing*, 28.-29. Juli, San Diego, USA, Proc. SPIE 3166, S. 21-30, 1997.
- [11] D. Musser & A. Stepanov. Algorithm oriented generic libraries. *Software - Practice and Experience*, 24(7): 623-642, 1994.

Parallel Global Optimization of High-Dimensional Problems

Siegfried Höfner, Torsten Schindler, and András Aszódi

Novartis Forschungsinstitut,

Brunnerstraße 59, A-1235 Vienna, Austria

{siegfried.hoefner,torsten.schindler,andras.aszodi}@pharma.novartis.com

<http://www.at.novartis.com/healthcare/nfi/default.asp>

Abstract. A parallel version of an optimization algorithm for arbitrary functions of arbitrary dimension N has been developed and tested on an IBM-Regatta HPC system equipped with 16 CPUs of Power4 type, each with 1.3 GHz clock frequency. The optimization algorithm follows a simplex-like stochastic search technique aimed at quasi-complete sampling of all the local minima. Parallel functionality is incorporated with the Message Passing Interface - MPI - version 1.2. The program is applied to typical problems of dimension $N=60$ and $N=512$ and the results are analyzed with respect to operability and parallel scalability.

1 Introduction

Many practical problems in science and technology are theoretically solvable but practically intractable because they require too long a computation. Computing time becomes a limiting factor particularly in mathematics and physics, where for certain problems the underlying basic principles defining the problem are well-known. However, they often lead to so many equations, or boost the dimensionality of a mathematical space so drastically, that a resulting computational procedure is hindered from successfully deriving a solution within an acceptable time frame. On the other hand supercomputers steadily increase in power and scope. Thus the aforementioned limitations may be overcome by high-performance computing systems and parallel architectures.

In this present study we investigate a general search algorithm that is usually applied to finding an optimum point - a minimum - on a highly complex functional surface. This minimum represents the behaviour of a particular N -dimensional scalar function $f_N(x_1, x_2, x_3, \dots, x_N)$ with respect to variations of the function's arguments $x_1, x_2, x_3, \dots, x_N$. Thus for the 2-dimensional case one could imagine a typical picture of a geological relief. A path in the (x_1, x_2) plane allows us to follow the rise and fall of the function's values on $f_2(x_1, x_2)$ the surface of the landscape. The goal of the search algorithm in such a case is to discover the positions of valleys and related valley base altitudes, finally picking out the optimal point with minimum geographical altitude. The problem with this approach immediately becomes clear even with this simple-minded 2-dimensional

example. It is difficult to determine the global optimal point because the algorithmic procedure should on the one hand try to detect valleys, but on the other hand also try to escape from these local valleys again once they are detected. This is because a local solution need not necessarily be a globally optimal point but the search for the latter is the ultimate goal of the algorithm. Furthermore, the situation becomes worse with increasing dimension N because the number of theoretical local minima grows exponentially also with N . For a real-world problem where N is of the order of 1000 or more the problem therefore becomes CPU bound.

In this paper we present a parallel application that performs the search for various local minima in parallel. In section 2 we outline the basic principles of direct search methods like the simplex optimization and their suitability to parallelization. The procedure is applied to a model test in section 3 and the results are analyzed in section 4.

2 Direct Search Methods – Simplex Optimization

One of the easiest methods to detect minima on a given functional surface is the simplex-method of Spendley, Hext and Himsworth [1] which was subsequently extended by Nelder and Mead [2]. The advantage of this particular method is that there is no need for gradient information, ∇f_N , of the function under consideration. As in a trial-and-error method one starts by setting up a set of $N+1$ non-identical points and computes corresponding functional values for these and subsequently rearranges all these $N+1$ positions in a well-defined way. The ultimate objective is to move the set of points closer and closer to the minimum. The idea of the Nelder and Mead algorithm is to successively improve the set of points by always searching for the least optimal point, i.e., the one with the highest functional value. This point is then moved using a geometrical transformation, specifically by a reflection through the centroid of all the remaining points in the simplex.

As pointed out by Torczon [3], the Nelder and Mead extension of the simplex-method exhibits the phenomenon of *restricted convergence*. This basically means that if the requested accuracy of the solution is too great the method fails to find the minimum. As a possible solution to this problem Torczon herself suggested the implementation of Parallel Multi-Directional Search (PMDS) [4]. Within the PMDS framework the way of improving the set of points forming the simplex is different to that of Nelder and Mead. The first major difference is the selection of the best adapted point for use as the centre of reflection, i.e., the point having the smallest functional value and therefore closest to the desired minimum. Secondly, all the remaining N points are then reflected through this selected centre, which is a process that may, in principle, be performed in parallel, hence the parallel character of this new simplex variant. However, according to Bassiri and Hutchinson [5], the phenomenon of *restricted convergence* was not eliminated by employing PMDS. This formed the motivation for the latter authors to devise an alternative method for performing the different moves during stepwise alter-

ation of the set of points which comprise the simplex. Note that another way of discriminating the various methods is to look at the number of points that are involved in the update of the simplex. Whereas in the early methods only one point was selected and became subject to geometric operations, in the second set of methods starting with PMDS, all points but one (the one with smallest functional value) were chosen and shifted around according to the employed set of geometric rules.

For the present purposes we will use the set of simplex-update-rules specific to the PMDS approach, which all are exhaustively defined from

$$p_i^*(x_1, x_2, x_3, \dots x_N) = (1 + t)p_l(x_1, x_2, x_3, \dots x_N) - tp_i(x_1, x_2, x_3, \dots x_N) \quad (1)$$

| update-rule | t-setting | application pre-condition |
|-------------|-------------|---|
| reflection | $t = 1$ | not converged, standard operation |
| expansion | $t > 1$ | one point of the reflected set gives rise to an even smaller f_N than current optimum |
| contraction | $0 < t < 1$ | reflection fails to produce any point with f_N smaller than the current optimum |

where p_i refers to one particular point of the $N+1$ points forming the simplex; p_l represents the only stationary point having the lowest functional value, and t is a parameter following the tabular description given above.

2.1 Enhancements towards Global Search

The procedure outlined in section 2 does not find a globally optimal point. It merely enables the local minimum lying next to the initial simplex to be discovered. Therefore additional features need to be established to enable the search for an overall optimal point. We suggest the following implementation:

- Use a regular simplex and standard simplex moves. Use the simplex moves only for translocation on the functional surface.
- At all the $N+1$ points defining a simplex employ independent local minimizations. The minimizations of these $N+1$ points are independent tasks and may be done in parallel.
- The resulting $N+1$ local minima are stored.
- Screen the list of detected local minima for unique entries as well as for new-coming entries to monitor the progress of the global search.
- Standard simplex-termination would occur when all the $N+1$ points define roughly the same position (a minimum - local or global). When this happens in the global search, all the $N+1$ points are re-initialized and the simplex starts afresh.
- The global search terminates if no new local minima are discovered even after several simplex re-initialization steps.
- Pick the point with smallest f_N from all the recorded unique local minima as the likely global minimum.

2.2 Parallel Aspects

From the profiling of a typical example with $N=60$ we could estimate the relative CPU-load for the isolated local minimization step to be of the order of 99.98 %. Thus within this context it was just a natural consequence to develop a parallelized version of the algorithm, described in section 2.1, that could perform the local minimization step at all the $N+1$ positions of the simplex in parallel. A straightforward solution was to split the $N+1$ local minimization steps into fractions, that could then be distributed over parallel operating CPUs.

The parallel algorithm is characterized as having very short communication intervals on the one hand and long lasting, independently operating parallel intervals on the other hand. The communication overhead is mainly due to sending the $N+1$ simplex coordinates to all participating parallel nodes and receiving the N coordinates of corresponding minima back from the parallel nodes. Compared to the time the individual nodes have to spend on computing their own sets of local minima this communication time is to a great extent negligible. The application inherent characteristics form the basis for an efficient parallel implementation using MPI. A master-node scheme was therefore adopted, where the master-process was responsible for all simplex-related organizational work, and the node-processes were due to local minimization jobs. Master "send" operations included sending of the $N+1$ points forming the current simplex to each of the nodes, while the nodes would just work on a predefined subset of these $N+1$ items and send back the computed corresponding local minima. Therefore there was no need to invoke higher level collective communication operations. We simply could get by with broadcasting buffered messages of length $N+1$ and receiving buffered messages of identical length via specific node-master sends.

For incorporation of parallel features the Message Passing Interface [6] seemed to be the most appropriate. This is because of the small additional programming efforts we had to undertake in obtaining a parallel version from modifying the already existing (and functional working) serial code. Furthermore, the target multiprocessor machine we had in mind for evaluation purposes was already equipped with a proprietary version of MPI-2 including FORTRAN 90 bindings (IBM parallel environment for AIX, version 3, release 2). Another important aspect when choosing MPI as the communication protocol was portability, since the described application is still being further developed and the number of available parallel architectures within our research institute is steadily growing. Regardless what parallel environment will finally become our target, the 16-CPU SMP IBM-machine appeared to better perform with MPI than PVM for comparable parallel tasks.

3 Results

A MPI-parallel version of the global simplex-like search algorithm described in section 2.1 was derived from the serial variant written in FORTRAN 90. A rather

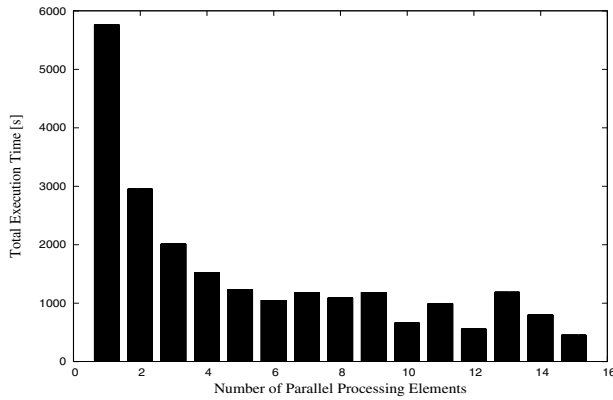


Fig. 1. Parallel performance of a simplex-like global search algorithm for a $N=60$ dimensional test function. This example is explored over 100 successive simplex cycles. The parallel architecture was an IBM-Regatta SMP system with 16 CPUs of type Power4 (1.3 GHz)

complicated test function [7],

$$f_N(\mathbf{x}) = \sum_{i=1}^{i=N/2} a x_{2i-1}^2 + b x_{2i}^2 + c \cos(\alpha x_{2i-1}) + d \cos(\gamma x_{2i}) - c - d \quad (2)$$

with $a = 1.0, b = 50.0, c = 3.0, d = 4.0, \alpha = 3\pi, \gamma = 4\pi$ has been used for all the test runs analyzed below.

At first the dimensionality was chosen to be $N=60$ and then a 100 cycle simplex-like global search procedure was invoked and the execution time measured on a single Power4 CPU (1.3 GHz) of a SMP 16-node IBM-Regatta server. The identical test calculation was repeated with successively increasing numbers of parallel processing elements (finally reaching 15 nodes). The master process itself accounted for a separate single CPU. A graphical representation of the results is given in Fig. 1. The resulting speed-up using 15 processors was 12.7-fold.

In addition a second run was set up where the dimensionality of the problem was increased to $N=512$ and the code executed on all available 16 CPUs of the parallel machine. Here the aim was to measure the number of newly-identified local minima in the course of the global search. Fig. 2 shows the percentage of the $N+1$ simplex points that led to newly identified local minima throughout the first 10 iteration steps.

4 Discussion

A typical simplex reflection move is demonstrated on a 2 dimensional example case in Fig. 3. As explained in section 2 there are 2+1 points involved and if

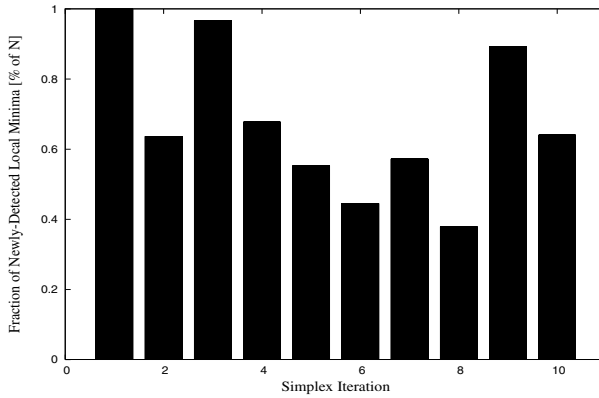


Fig. 2. Rate of de-novo identification of local minima in the course of iterative simplex moves with following local minimizations for a $N=512$ dimensional test case

we follow the PMDS-strategy according to Torczon we move all points but one. The point that remains unchanged is the one having smallest functional value f_2 from the first, which is the boundary point of the two triangles in Fig. 3. Next we apply local minimization steps at all the various simplex positions. This leads us to the closest local minimum, which is illustrated by the zigzag-pathways in the contour plot of Fig. 3. This must be considered as the major difference to other existing global search methods. First of all we do not feed back the information of positions of local minima to the subsequent simplex-moves. The detected local minima are just stored and screened for new values. This gives us an idea of the progress of the global optimization process. Secondly, we perform the local minimization jobs, which all are independent tasks, in parallel. In particular we apply a minimization procedure similar to the one outlined in [8]. The important thing to realize with this method is that it is a gradient-free technique and that it employs the well-known Powell method of local minimization.

In the beginning of the procedure and when all the local minima coincide we initialize the simplex. For this purpose we define one point randomly. Based on this random point a new regular simplex is formed with randomly chosen edge-length.

Parallel scaling exhibits a non-optimal behaviour with large numbers of CPUs. For example as seen from Fig. 1 increasing the number of CPUs from 10 to 11 actually results in slow-down instead of speed-up. However this is due to the fact that we have chosen a 60-dimensional case. There is no optimal way of splitting a group of 60 elements into equal-sized portions of 11 fractions. So for example in such a case we have to employ 8 CPUs with 5 minimizations, and always 1 CPU with 6,7 and 8 minimizations, which is certainly shifting the bottleneck to the process dealing with 8 minimizations. Therefore the best parallel performance is observed in situations where 60 may be divided through the num-

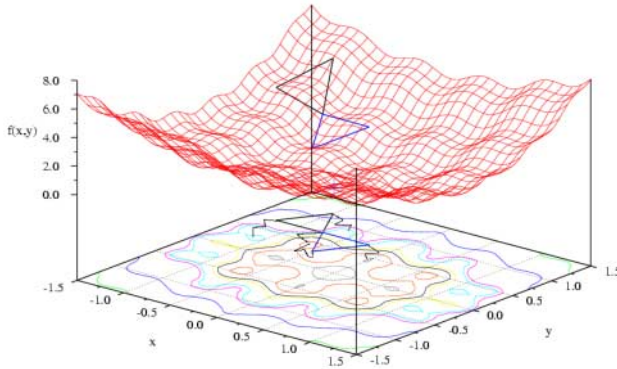


Fig. 3. Simplex reflection move illustrated with a 2 dimensional example. From the 2+1 simplex points the one with smallest f_2 is held constant, while the remainder are reflected through this very point. In the contour plot we also indicate the following local minimization steps

ber of CPUs (see Fig. 1). Furthermore according to Amdahl's Law, with a serial fraction of 0.0002, we should see a much higher degree in parallel speed-up. For example using 12 CPUs we should theoretically obtain a factor of 11.97x but see 10.32x instead. The reason for this is that different local minimization steps may last differently long depending on how far away the point we start with is actually located from the next local minimum.

There is a characteristic drop in the number of newly detected local minima every second step of the simplex moves (see Fig. 2). This is the effect of closer related simplex-point-positions for expansion and contraction moves which both follow the standard reflection move. So what we see in Fig. 2 is the effect of alternating reflection and contraction/expansion moves from which only the first type always leads to a completely different set of positions (also compare to Fig. 3). In addition we report that the N=512 example needed more than 3 full 16xCPU-days to complete the initial 10 iterations which is far from exhaustive exploration of the entire functional surface.

5 Conclusion

A new global search algorithm has been presented here in which "simplex-like" moves are followed by local minimization steps. The advantage with this new method is that a robust and well-established technique (simplex) is combined with additional local minimization-refinements that may be efficiently performed in parallel. While the current implementation works on a mathematically well defined analytical test-function, the concept may be easily generalized to ex-

plore any arbitrarily complex potential surface. Such minimization problems are often encountered in rational drug design, in particular in the analysis of conformational preferences of small molecules. Another possible application area is the computational study of ligand-receptor interactions (docking) that has the potential of speeding up the lead optimization stage of the drug discovery process. Work is currently underway in our laboratory to explore these application opportunities where the speed-up provided by parallelization offers a significant methodological advantage.

Acknowledgment

The authors would like to thank Dr. Adrienne James for helpful discussions and Dr. Pascal Afflard and his colleagues at Novartis Basel for granting access to their computing facilities.

References

- [1] Spendley, W., Hext, G. B., Himsforth, F. R.: Sequential application of simplex design in optimisation and evolutionising operation. *Technometrics* **4** (1962) 441 149
- [2] Nelder, J. A., Mead, R.: A simplex method for function minimization. *Comp. J.* (1963) 308 149
- [3] Torczon, J. V.: On the convergence of the multi-directional search algorithm. *SIAM J. Optimization* **1** (1991) 123–145 149
- [4] Torczon, J. V.: Multi-Directional Search: A Search Algorithm for Parallel Machines. PhD thesis, Rice University, Houston, Texas (May 1989) 149
- [5] Bassiri, K., Hutchinson, D.: New Parallel Variants on Parallel Multi-Dimensional Search for Unconstraint Optimisation. research report 94.28, Div. of Computer Science, University of Leeds, UK, (1994) 149
- [6] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. 2nd edition. MIT Press, Cambridge, MA, (1999) 151
- [7] Dennis, S., Vajda, S.: Semiglobal Simplex Optimization and its Application to Determining the Preferred Solvation Sites of Proteins. *J. Comp. Chem.* **23**, 2 (2002) 319–334 152
- [8] Brent, R. P.: Algorithms for Minimization Without Derivatives, Chapter 7. Dover Publications, Mineola, New York, 0-486-41998-3, (2002) 153

Adjusting the Lengths of Time Slices when Scheduling PVM Jobs with High Memory Requirements^{*}

Francesc Giné¹, Francesc Solsona¹, Porfidio Hernández², and Emilio Luque²

¹ Departamento de Informàtica e Ingenieria Industrial, Universitat de Lleida, Spain
`{sisco, francesc}@eup.udl.es`

² Departamento de Informàtica, Universitat Autònoma de Barcelona, Spain
`{p.hernandez, e.luque}@cc.uab.es`

Abstract. Our research is focussed on keeping both local and parallel jobs together in a non-dedicated cluster and scheduling them efficiently. In such systems, the overflow of the physical memory into the virtual memory usually causes a severe performance penalty for distributed jobs. This impact can be reduced by means of giving more time slice length to parallel tasks in order better to exploit their memory reference locality. Thus, an algorithm is presented to adjust the length of the time slice dynamically to the necessity of the distributed and local tasks. It is implemented in a Linux cluster and evaluated with PVM jobs.

1 Introduction

The challenge of exploiting underloaded workstations in a NOW for hosting parallel computation has led researchers to develop different techniques in an attempt to adapt the traditional uniprocessor time-shared scheduler to the new situation of mixing local and distributed workloads. In such systems, the PVM framework has allowed the development of efficient distributed applications.

The performance of PVM applications can drastically decrease if memory requirements are not kept in mind [5, 2, 3]. In a non-dedicated system, the dynamic behavior of local users or a distributed job mapping policy without memory considerations cannot guarantee that parallel jobs have as much resident memory as would be desirable throughout their execution. In these conditions, the local scheduler must coexist with the operating system's demand-paging virtual memory mechanism. Thus, in distributed environments, the traditional benefits that paging provides on uniprocessors may be degraded due to two factors: the high overhead of page faults caused by synchronization delays and the pernicious reference patterns of these parallel applications [2]. The reduction of the impact of demand paged virtual memory across a non-dedicated cluster is

^{*} This work was supported by the MCyT under contract TIC 2001-2592 and partially supported by the Generalitat de Catalunya -Grup de Recerca Consolidat 2001SGR-00218.

the main aim of this paper. It is achieved by means of giving more time slice (or quantum) length to parallel jobs in order better to exploit their memory reference locality. However, an excessively long time slice could degrade the performance of fine grained applications [8] and the response time of interactive applications. Thus, the proper length of time slice should be set according to the characteristics of concurrent jobs and this changes as jobs and/or memory requirements vary.

A new algorithm is presented and discussed in this paper to adjust dynamically the time slice length of every local scheduler in a non-dedicated NOW according to local user interactivity, the memory requirements and communication behavior of each parallel job. This technique is implemented in a Linux NOW and is compared with other alternatives by means of PVM applications.

The rest of the paper is organized as follows. In section 2, the proposed algorithm and its implementation are explained. Its performance is evaluated and compared in section 3. Finally, conclusions are detailed.

2 DYNAMICQ: An Algorithm to Adjust the Time Slice

An algorithm –denoted as *DYNAMICQ* (for Dynamic Quantum)– to compute the time slice in each node is proposed and discussed. The following subsection gives an overview of the notation and main assumptions taken into account.

2.1 Assumptions and Notation

Our framework is a non-dedicated cluster, where all nodes are under the control of our scheduling scheme. Every local scheduler takes autonomous decisions based on local information provided by the underlying operating system. Some of the following assumptions are made taking Linux properties [1] into account to make the subsequent implementation of the algorithm easier.

A time sharing scheduler is assumed with process preemption based on ranking processes according to their priority. The scheduler works by dividing the CPU time into *epochs*. In a single epoch, each process (*task*) is assigned a specified time slice, denoted as $task.q_n$ (task's time slice for the n^{th} epoch), which it is allowed to run. When the running process has expired its time slice or is blocked waiting for an event, another process is selected to run from the Ready Queue (RQ) and the original process is made to wait. The epoch ends when all the processes in the RQ have exhausted their time slice. The next epoch begins when the scheduler assigns a fresh time slice to all existing processes.

The Memory Management Unit (MMU) provides demand-paging virtual memory. This means that if the page referenced by a task is not in its *resident set* (allocated pages in the main memory), a *page fault* will occur. This fault will suspend the task until the missing page is loaded in its resident set. It is assumed that the operating system maintains information about the page fault rate ($task.flt$) produced by each task.

When the kernel is dangerously low on memory, swapping is performed. A swapping mechanism (Linux v.2.2.15) is assumed in which the process with

most pages mapped in memory is chosen for swapping. From the located process, the pages to be swapped out are selected according to the *clock replacement algorithm*, an approximation to the *Least Recently Used (LRU) algorithm*.

Taking the previous assumptions into account, the length of the time slice should be computed according to the following aims:

1. The synchronization inherent in message-passing applications makes page faults much more expensive than on a uniprocessor, given that each page fault can cause cascading delay on other nodes. Thus, the *minimization of the page faults* in every node will have benefits for the global performance of the cluster. This is the main aim of the *DYNAMICQ* algorithm.
2. The *coscheduling of distributed jobs* according to their own communication needs in order to minimize synchronization/communication waiting time between remote processes. With this aim, it is assumed that every local scheduler applies a coscheduling technique which consists of giving more scheduling priority to tasks with higher receive-send communication rates. This technique, named *predictive coscheduling*, is based on the assumption that high receive-send message frequencies imply that potential coscheduling with remote processes is met and an accurate selection of the correspondents – the most recently communicated – processes in each node can be made. This technique has been chosen because of the good performance achieved in a non-dedicated PVM-Linux NOW [6] with respect to other coscheduling techniques, such as *dynamic* [8] or *implicit* coscheduling [7].
3. A *good response time* of interactive tasks belonging to local users must be guaranteed.

2.2 DYNAMICQ: A Dynamic Quantum Algorithm

Fig. 1 shows the steps for calculating the time slice taking the preceding aims into account. The *DYNAMICQ* algorithm is implemented inside a generic routine called *ASSIGN_QUANTUM*, which assigns a new time slice to all the existing processes in the node, which are arranged by means of a circular list. It is computed by every local scheduler every time that a new epoch begins. *DYNAMICQ* can work in three different manners according to the memory requirements (*SWAPPING* condition) and local user logging (*LOCAL_USER* condition):

- *No swapping*: In this case, the algorithm assigns the default Linux time slice¹ (DEF_QUANT) to all tasks in order to preserve the performance of the local user.
- *Swapping with a local user*: A small time slice must be set bearing in mind that the response time of interactive tasks must be preserved. However, a slight increase in the time slice length can visibly contribute to reducing the number of page faults since each task can fit its working set² in the main

¹ It is 200ms.

² Many distributed applications touch a large fraction of their total data set for relatively small time interval (around 100ms)[2].

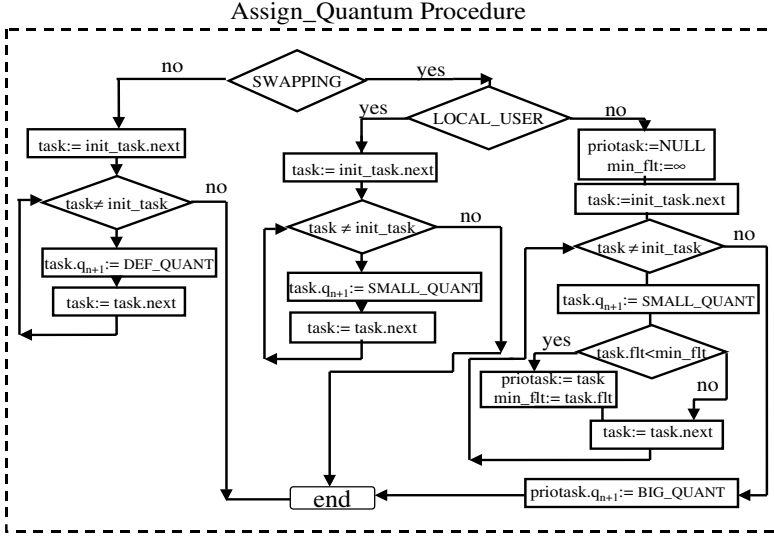


Fig. 1. Flow graph of the DYNAMICQ algorithm

memory better. This way, it can improve the distributed task performance. Thus, a compromise between both needs must be reached. The new time slice assigned to all tasks –denoted as *SMALL_QUANT*– is set to double the default time slice. Experimentally, we will prove that with this new time slice the number of page faults can be reduced by half in some cases, while a good response time is maintained for local tasks.

- *Swapping without local users:* A *SMALL_QUANT* is assigned to all the existing tasks, except the distributed task with the lowest page fault rate ($task.flt < min_flt$), to which a new time slice (*BIG_QUANT*) is assigned equal to $8 * DEF_QUANT$. It will let such distributed tasks (*priortask*) finish as soon as possible. Thus, on its completion, the released memory will be available for the remaining distributed applications. Consequently, major opportunities arise to advance execution of all the remaining tasks. This idea was proved by simulation in [3] with excellent results. Finally, it is worthwhile pointing out that a time slice longer than this could lead to benefits reached by the minimization of the page faults not compensating for the slowdown introduced by the loss of coscheduling. Given that the assumed *predictive* coscheduling does not apply a global control to schedule all the processes of a parallel job concurrently, a situation in which scheduled processes that constitute different parallel jobs contend for scheduling their respective correspondents could occur quite frequently. Thus, if the time slice is too long, the context switch request through sent/received messages could be discarded and hence the parallel job would eventually be stalled until a new context-switch was initiated by the scheduler.

2.3 Implementing DYNAMICQ Inside the Linux Kernel

DYNAMICQ was implemented in the Linux Kernel v.2.2.15 [1]. The implementation in the kernel space gives transparency to the overall system and allows its application regardless of the message passing environment (PVM, MPI, etc ...). The drawback is in portability. A patch, with the following modifications, must be introduced into each cluster node in the Linux source:

Local User Activity: Linux treats I/O devices as files. So, the same file's structures are used to maintain all the necessary information to handle an I/O device. Specifically, we used the *inode structure* associated with the keyboard file `"/dev/stdin"` and the mouse file `"/dev/mouse"` to track user activity. Thus, at the beginning of every epoch, the access time (*i_atime* field of inode structure) to the keyboard and mouse file is checked, setting a new kernel variable –denoted as *LOCAL_USER*– to False, when there is no interactivity user for 1 minute or more, or to True when there is one.

Page Fault Rate: In Linux, the *task struct* associated with each task contains a field (*maj_flt*) which counts the page faults with access to the disk of such a task from the beginning of its execution. Given that we are interested in obtaining these page faults during every epoch, a new field (*flt*) –which is recomputed at the beginning of every epoch– has been added to the *task struct*.

Time Slice Length: Each process has a *base time slice* (static priority), which is inherited from its parent. In order to select a process to run, the scheduler must consider the *dynamic priority* of each process, which is the sum of the base time slice and the number of ticks of CPU time left to the process before its time slice expires in the current epoch. Whenever an epoch finishes, the dynamic priority is recomputed. Thus, our implementation involved the modification of the *base time slice* according to the DYNAMICQ algorithm.

Predictive Coscheduling: In order to achieve coscheduling, whenever a task is inserted into the RQ, its dynamic execution priority is increased according to its current communication rate ($rq + sq$), where *rq* (*sq*) is the number of packets in the receive (send) socket packet-buffering queue. So, the task with most pending receiving/sending messages will have the highest dynamic priority. Thus, the scheduler, which is invoked every time that a process is woken up, will schedule this task if it has a higher dynamic priority than the current scheduled task. Coscheduling is thus achieved. With this aim, we implemented a function to collect the sending/receiving messages from the socket queues in the kernel. In [6], the reader can find a detailed explanation of the predictive coscheduling implementation in Linux.

3 Experimentation

The experimental environment was composed of eight 350MHz Pentium II with 128MB of memory connected through a Fast Ethernet network. The performance of the distributed applications was evaluated by running three PVM

Table 1. Workloads. Local(z - y/x) means that one local task of yMB of memory and $x\%$ of CPU requirements has been executed in z different nodes

| | Workload (Wrk) | Cluster State/mem_load |
|---|-----------------------------------|---------------------------------------|
| 1 | IS.A (101s)+MG.A(105s)+MG.B(265s) | All nodes with memory overloaded/121% |
| 2 | MG.A+SP.A(250s)+MG.B | All nodes with memory overloaded/135% |
| 3 | MG.B+local(4-50/50%) | 4 nodes with memory overloaded/83% |
| 4 | IS.A+SP.A+local(4-50/90%) | 4 nodes with memory overloaded/91% |

distributed applications from the NAS suite with class A and B: *IS* (a communication bound), *MG* (a computation bound) and *SP* (computation and communication are balanced). One task of each job was mapped in every node. The local workload was carried out by means of running one synthetic benchmark, called *local*. It allows the CPU activity to alternate with interactive activity. The CPU is loaded by performing floating point operations over an array with a size and during a time interval set by the user (in terms of time rate). Interactivity was simulated by means of running several system calls with an exponential distribution frequency (mean=500ms by default) and different data transferred to memory with a size chosen randomly by means of a uniform distribution in the range [1MB,...,10MB]. At the end of its execution, the benchmark returns the system call latency and wall-clock execution time.

Three environments were evaluated: the plain Linux scheduler (denoted as *LINUX*), the DYNAMICQ algorithm and the *STATICQ* mode. In the *STATICQ* mode, all the existing tasks in each node are assigned the same time slice, which is set according to the value passed from a system call implemented by us. In such environments, four workloads –two dedicated and other two non-dedicated– with different memory and communication requirements were executed. Table 1 shows the memory state of every node when every workload was running in the cluster together with the mean memory load per node (*mem_load*), which is the sum of the memory requirements of all tasks belonging to one node with respect to its main memory size. It is averaged over all the nodes. Additionally, table 1 shows the execution time of every distributed benchmark (between brackets) when it was executed on a dedicated system.

The performance of the parallel/local jobs was validated by means of two different metrics³: **Mean Page Faults (MPF)** (defined as the page faults experimented by a task averaged over all the tasks of the cluster) and **Slowdown** (the response-time ratio of a distributed/local job on a non dedicated system in relation to the time taken on a system dedicated solely to this job. It was averaged over all the distributed/local jobs of the cluster). Both parameters were obtained by means of the *MemTo tool* [4].

Fig. 2 shows the MPF (left) and Slowdown (right) metrics for all the workloads while varying the time slice length from 200ms to 25.6s in the *STATICQ* mode. The analysis of MPF for Wrk2 shows that the number of page faults

³ The obtained metrics are the average of 10 different executions.

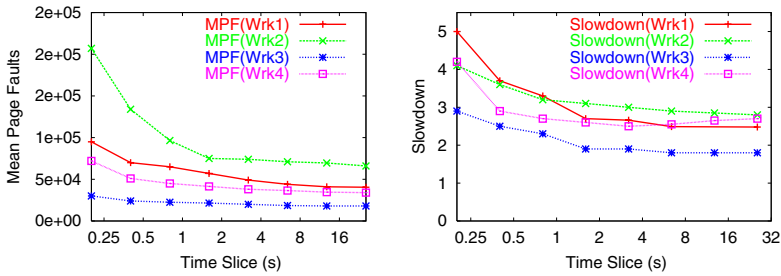


Fig. 2. STATICQ mode. MPF (left) and Slowdown (right) metrics for PVM jobs

falls drastically when the time slice is increased to $1.6s$, whereas it becomes almost stable with a time slice over $1.6s$. This means that Wrk2 does not need a time slice longer than $1.6s$ to fit its whole working set⁴ in the main memory. In the other cases, the MPF obtained decreases more gradually as a consequence of smaller memory requirements than in the Wrk2 case. The MPF behavior is reflected in a better Slowdown (see fig. 2 (right)). It is worth pointing out that this improvement is greater in the Wrk1 and Wrk4 cases because the IS benchmark has extreme communication and synchronization requirements and thus it is more sensitive to the diminishing of the page faults. Note that the Wrk4 Slowdown is slightly increased with a time slice over $3.2s$ due to a loss of coscheduling.

Fig. 3 shows MPF (left) and Slowdown (centre) metrics for all the workloads under all the environments (LINUX, DYNAMICQ and STATICQ with a time slice of $1.6s$). As expected, the worst performance is obtained with the plain LINUX. In general, DYNAMICQ and STATICQ obtain similar results in the Wrk2 and Wrk3 cases, whereas DYNAMICQ obtains the best results in the Wrk1 and Wrk4 cases. This is due to the fact that DYNAMICQ gives priority to the task with the lowest page fault rate (IS benchmark in the Wrk1 and Wrk4 cases), which normally has the shortest execution time [3] and thus, the memory resources are free as soon as possible. Additionally, fig. 3 (right) shows the overhead introduced by Wrk3 and Wrk4 into the *local* task under the three environments when the CPU requirements of *local task* were decreased from 90% to 10% (in this case the slowdown basically reflects the overhead added to the sys-call latency). It can be seen that the results obtained with Linux are slightly better than those with DYNAMICQ. On the other hand, STATICQ obtains the worst results. This behavior is because the STATICQ and DYNAMICQ modes give more execution priority to distributed tasks with high communication rates, thus delaying the scheduling of local tasks until distributed tasks finish their time slice. This priority increase has little effect on local tasks with high CPU requirements but provokes an overhead that is proportional to the time slice length in the response time of interactive tasks (CPU requirements of 10%).

⁴ The MG and IS benchmarks's working sets are shown in [4].

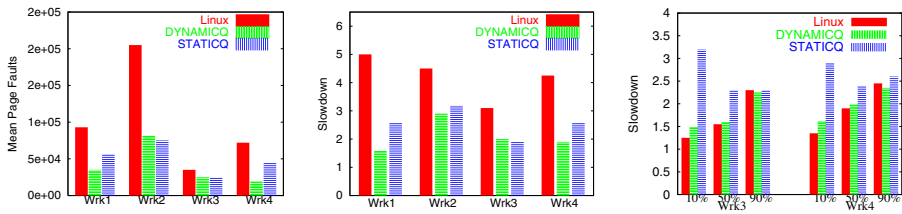


Fig. 3. MPF (left), Slowdown(PVM) (centre) and Slowdown(local) (right)

This effect is reflected in the high slowdown introduced by the STATICQ mode when *local task* has low CPU requirements but high interactivity needs.

4 Conclusions

Demand-paged virtual memory benefits, which are well known for uniprocessors, are not so clear for NOW environments due to the high overhead of page faults caused by synchronization delays. In this paper, we prove experimentally that the number of page faults across a non-dedicated cluster can be reduced when the time slice length is increased. However, it can damage the performance of interactive tasks and coscheduling techniques applied to distributed tasks. So, in order to preserve a good response time for interactive tasks and maintain the coscheduling between distributed tasks, an algorithm is presented to adjust the length of the time slice dynamically to the needs of the distributed and local tasks. It is implemented in a Linux cluster and is evaluated by means of PVM jobs. The obtained results have demonstrated its good behavior in reducing the number of fault pages and the Slowdown of distributed tasks and maintaining the performance of local tasks with respect to other policies.

Future work will be directed towards extending our analysis to a wider range of workloads, including new parallel paradigms and a different kind of local user.

References

- [1] D. Bovet and M. Cesati. "Understanding the Linux Kernel". *O'Reilly*, 2001. 157, 160
- [2] D. Burger, R. Hyder, B. Miller and D. Wood. "Paging Tradeoffs in Distributed Shared-Memory Multiprocessors". *Journal of Supercomputing*, vol. 10, 1996. 156, 158
- [3] F. Giné, F. Solsona, P. Hernández and E. Luque. "Coscheduling Under Memory Constraints in a NOW Environment". *7th Workshop on Job Scheduling Strategies for Parallel Processing*, LNCS, Vol. 2221, 2001. 156, 159, 162
- [4] F. Giné, F. Solsona, X. Navarro, P. Hernández and E. Luque. "MemTo: A Memory Monitoring Tool for a Linux Cluster." *EuroPVM/MPI'2001*, LNCS, vol.2131, 2001. 161, 162
- [5] S. Setia, M. S. Squillante and V. K. Naik. "The Impact of Job Memory Requirements on Gang-Scheduling Performance". *In Performance Evaluation Review*, 1999. 156

- [6] F. Solsona, F. Giné, P. Hernández and E. Luque. "Predictive Coscheduling Implementation in a non-dedicated Linux Cluster". *EuroPar'2001*, LNCS, vol.2150, 2001. [158](#), [160](#)
- [7] F. Solsona, F. Giné, P. Hernández and E. Luque. "Implementing Explicit and Implicit Coscheduling in a PVM Environment". *EuroPar' 2000*, LNCS, vol.1900, 2000. [158](#)
- [8] A. Yoo and M. Jette. "An Efficient and Scalable Coscheduling Technique for Large Symmetric Multiprocessors Clusters". LNCS, vol.2221, [157](#), [158](#) 2001.

A PVM-Based Parallel Implementation of the REYES Image Rendering Architecture

Oscar Lazzarino, Andrea Sanna, Claudio Zunino, and Fabrizio Lamberti

Dipartimento di Automatica e Informatica
Politecnico di Torino Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy
`oscar.lazzarino@tiscali.it`
`{sanna,c.zunino,lamberti}@polito.it`

Abstract. In this paper a PVM-based distributed platform of the well known REYES rendering architecture developed by Pixar is presented. This work effectively tackles issues related to load balancing and memory allocation by a master-slave paradigm. In particular, the rendering is distributed performing both an image and an object space subdivision; in this way, low memory resources are necessary to the slave side. Examples show the effectiveness of the proposed work.

1 Introduction

Several algorithms are able to produce very realistic synthetic images, but their main drawbacks are the high computational times and the memory requirements. Although CPUs have enormously increased their computational capacity and the cost of memory chips has decreased, single processor solutions are often not able to provide the rendering of a scene within acceptable times. On the other hand, distributed low-cost architectures based on PC clusters have been proved to be effective and performing solutions.

This paper proposes a distributed implementation based on PVM [1] of the well-known REYES architecture [2] developed by Pixar. Although REYES is not able to directly take into account global illumination effects, reflection and transparency can be simulated in order to obtain a high level of realism.

Two main issues have to be addressed: load balancing and memory occupation. The first issue can be addressed performing a dynamic load balancing, that is, the image is split into a set of regions dynamically assigned to computation nodes (image space subdivision). An effective memory usage can be obtained dynamically providing every node only of primitives strictly necessary to render the region of image under analysis (object space subdivision).

The proposed solution performs a hybrid solution that merges image and object space subdivision, providing both an effective dynamic load balancing by means of a master-slave paradigm and an efficient memory management that avoids the duplication of the entire database on every slave node.

The paper is organized as follows: Section 2 reviews the main approaches involved in distributed and parallel rendering, while Section 3 briefly presents the

sequential version of the algorithm considered in this work. Section 4 describes the details of the parallel implementation and Section 5 shows results obtained testing the proposed architecture over a Gigabit-Ethernet PC cluster.

2 Background

High quality rendering algorithms always need high computational capacity and parallel approaches have been often used in order to reduce rendering times.

Independently of the rendering algorithm, two main strategies can be identified: image space subdivision-based and object space subdivision-based.

Image space subdivision-based algorithms split the image plane into several regions, and each processor has to compute one or more of them. With this approach the load distribution can be either static or dynamic; in the first case, each processor has a priori knowledge of pixels to be rendered, while, in the second case, the load is assigned at run time. Image space subdivision-based algorithms may suffer from two main drawbacks: the implementation may be critical on shared-memory machines, since a large number of accesses to memory may become a bottleneck (in massively parallel computation), and the whole scene database has to be duplicated at each processor when distributed-memory machines are employed.

The object space subdivision-based methods were designed to address the problem of the implementation of parallel algorithms on distributed memory machines. In such computers each processor has a small amount of local memory, so that each processor has to load into its memory a small subset of the entire database. Each processor checks the objects stored in its local memory and results are sent to the other processors by messages. The strategy of the database subdivision is very important because it affects the workload distribution among processors; the object distribution can be either static or dynamic. In the former case, each processor has a priori knowledge of the elements to be loaded into its memory, while in the latter case the objects are assigned at run time.

Indeed, a lot of work is known in the literature concerning the parallelization of ray-tracing and radiosity algorithms.

Ray-tracing is able to provide photo-realistic rendering by considering global illumination effects such as transparency and reflection. Moreover, the computation of a pixel is independent of the others, therefore, a straightforward parallelization can be obtained [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Of particular interest are the distributed rendering algorithms tailored for PC/workstation clusters. For instance, [13] and [14] proposed two distributed implementations of the well-known POV-Ray ray tracer [15] based on PVM.

Radiosity methods produce highly realistic illumination of closed spaces by computing the transfer of light energy among all of the surfaces in the environment. Radiosity is an illumination technique, rather than a complete rendering method. However, radiosity methods are among the most computationally intensive procedures in computer graphics, making them an obvious candidate for

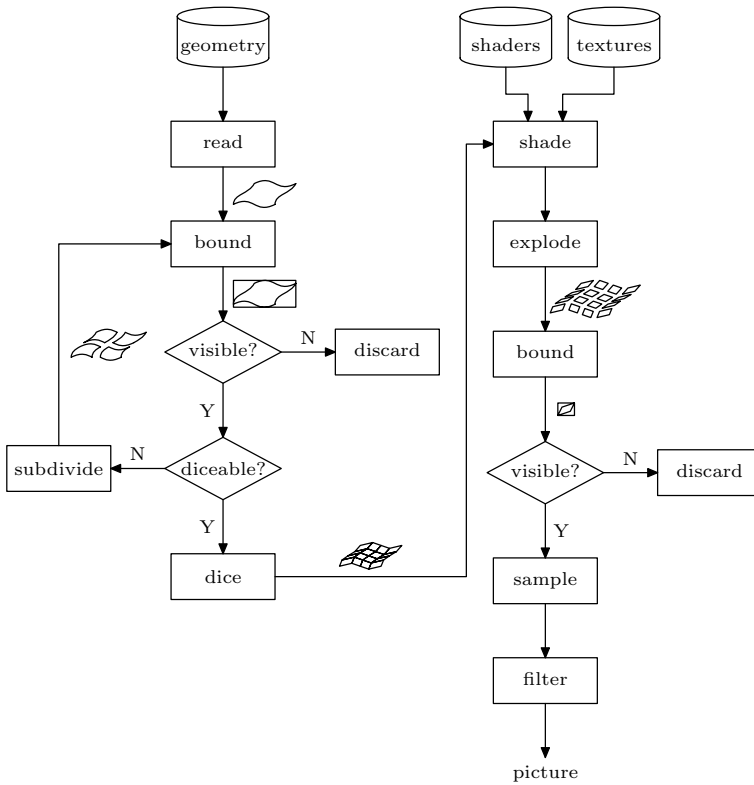


Fig. 1. Flow of data in the REYES pipeline

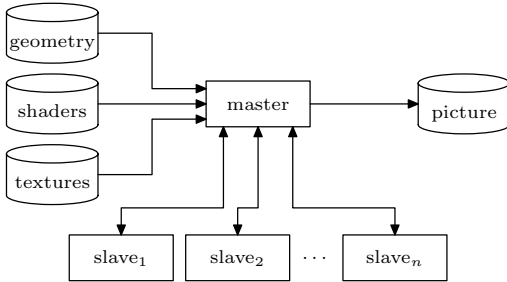
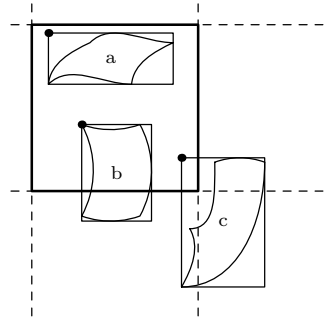
parallel processing. Many of the parallel radiosity methods described in the literature attempt to speed up the progressive refinement process by computing energy transfers from several shooting patches in parallel (i.e., several iterations are performed simultaneously) [16, 17, 18].

At our knowledge, a new proprietary parallel version of the REYES architecture will be included in Pixar's RenderMan[®] R11 which will be released in Q4, 2002.

3 The REYES Architecture

The REYES architecture can be viewed as a pipeline through which the data flows independently and therefore it can be classified as a local illumination system. This fact allows the rendering of scenes of arbitrary complexity; global effects as shadows, reflections, and refractions cannot directly be computed by the algorithm, but various methods exist to overcome this restriction.

The design principles behind REYES are discussed in depth in [2]; in this section, a brief outline of the algorithm will be only described. The algorithm is

**Fig. 2.** Master-slave architecture**Fig. 3.** Buckets subdivision

summarized in Fig. 1. The algorithm uses a z-buffer to remove hidden surfaces. The primitives in the input file are read sequentially and for each one a bounding box is built. This bounding box is used to determine whether the primitive is visible or not. If the bounds are completely outside the viewing frustum, the primitive is culled.

After these preliminary checks, the primitive is repeatedly split until it can be safely *diced* into a grid of *micro-polygons*. A micro-polygon is a little (not bigger than a pixel) quadrilateral polygon. The dicing operation is performed when the evaluation of the resulting grid shows it will not contain too many micro-polygons and their size will be not too different.

The grid is then shaded and broken into independent micro-polygons. Every single micro-polygon is then bounded, and if the bounding box happens to be entirely out of the the viewing frustum, the micro-polygon is discarded. Visible micro-polygons are then sampled one or more times per pixel, and the samples are checked against the z-buffer.

When there is no more data in the input file, the samples are then filtered to remove aliasing artifacts, dithered, and quantized and the final image is produced.

4 Parallel Implementation

The proposed algorithm is based on the REYES architecture described in Section 3. To design a distributed rendering environment, the original architecture has been modified following a master-slave approach as shown in Fig. 2. The system consists of a master process and of several slave processes which actually implement the REYES algorithm on a subset of the input data. It is up to the master to split the original data set into smaller subsets and to dispatch them to slave processes in order to efficiently distribute the computational load among available cluster nodes. This goal is achieved dividing the image area in rectangular blocks, named *buckets*.

The geometric primitives generating the scene are assigned to the bucket containing the top-left corner of the primitive bounding box, as shown in Fig. 3.

Buckets are then submitted to slave processes where the rendering of the corresponding image block takes place. It has to be remarked that this bucket selection criteria assigns primitives covering adjacent blocks to a single bucket thus forcing the designated slave to process data which could be outside of its scope. It will be shown how the slave identifies those primitives falling out of the block and passes them back to the master process that will dispatch them to the proper slave.

4.1 Distributed Algorithm Description

In the following, the steps of the parallel rendering algorithm are presented and PVM master-slave message passing details are outlined:

- M→S** the master parses the input file describing the scene and performs primitive-to-bucket assignments; then the master runs the slave processes and sends them an initialization message `INIT` containing camera, image and rendering algorithm related information including camera position and orientation, image resolution, buckets size, number of samples per pixel, micro-polygons grid size, shading rate, reconstruction filter, quantization, dithering and clipping planes parameters;
- M←S** when the slave is ready to process a bucket, it sends an `ASK_BUCKET` to the master;
- M→S** the master selects a bucket from the scheduling queue according to the priority parameter (see section 4.2) and sends to the slave a `BUCKET` message containing bucket coordinates;
- M←S** the slave performs several initialization operations (i.e. clears the z-buffer and the frame buffer) and sends the master an `ASK_PRIMS` message to request primitives;
- M→S** the master sends the slave a `PRIMS` message containing a set of primitives and micro-polygons for the current bucket in a serialized format (in this way, both an image and an object space subdivision is performed);
- M←S** the slave processes primitives and micro-polygons according to REYES algorithm. As previously mentioned, the REYES split operation can generate primitives falling outside the bucket bounds. These primitives are stored in a primitives temporary queue. Primitives that pass the diceable-test are diced into a micro-polygons grid and shaded. Micro-polygons corresponding to primitives partially outside bucket bounds are stored in a micro-polygons temporary queue. When the size of the primitives or micro-polygons queues reach a particular threshold value, a `JUNK` message containing all the data in the temporary queues is sent to the master. The master delivers received data to the appropriate buckets, according to the new bounding boxes;
- M↔S** when the primitives have been processed, the slave sends additional `ASK_PRIMS` messages to the master. The master replies with additional `PRIMS` messages and sends an `EOB` (End of Bucket) when there is no more data in the bucket;

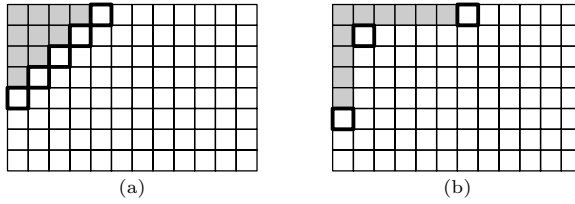


Fig. 4. Buckets to slave scheduling: the gray buckets are completed, while highlighted ones are ready to be scheduled

M←**S** the slave performs sample filtering operations, as well as gamma correction, dithering and quantization, and sends the master a **RESULT** message containing the processed image portion. Moreover, an **ASK_BUCKET** message notifies the master that the slave is ready to receive a new job;

M→**S** the master can reply by sending a new bucket or by repeating the **EOB** message indicating all buckets have been processed and the slave can stop its execution.

Slave processes are started and terminated by the master and only master-slave communications are allowed. It is worth mentioning that the master is the only process needing a direct access to input data (i.e. scene description, shaders and, textures), as shown in Fig. 2.

4.2 Load Distribution

A key aspect of the proposed parallel rendering architecture is the job distribution strategy. The choice of the bucket-to-slave allocation algorithm deeply influences performances. This is mainly due to the fact that, while processing a bucket, the slave can encounter primitives which should be assigned to adjacent blocks. This flow of primitives between neighboring buckets always proceeds in the right-to-left and up-to-down directions, as clearly shown in Fig 3. Therefore, bucket dispatch to slave processes must follow well-defined scheduling rules. In particular, a bucket $B_{i,j}$ is ready to be scheduled only if both buckets $B_{i-1,j}$ and $B_{i,j-1}$ have been completely processed. When the master is started, only bucket $B_{0,0}$ is ready and it is assigned to a slave. As new buckets become ready, the master inserts them into a scheduling queue. The best performance can be obtained by trying to maximize the number of ready buckets.

It can be shown that this goal can be achieved by assigning to each bucket a queue extraction priority given by the sum of row and column indexes, thus forcing the scheduling algorithm to proceed in a diagonal direction. Fig. 4(a) shows that, by adopting the optimal scheduling scheme, when ten buckets have been processed, there are five buckets ready to be dispatched to slave processes. In Fig. 4(b) a sub-optimal scheduling scheme is presented: as in Fig. 4(a), ten buckets have already been processed but there are only three ready buckets.



Fig. 5. Three test scenes

The optimal case is not always possible, since the buckets can require different amount of time to be computed. Nevertheless, with the adopted strategy, the master manages to keep all the slaves busy for most of the time.

5 Examples and Remarks

The system has been tested with three different input scenes, chosen for their characteristics. The “frog scene” contains many geometric primitives, but most of them are small in size. This implies that most primitives are entirely enclosed in a single bucket, and so the traffic of discarded primitives and micro-polygons is quite reduced. The “teapot scene” is constituted of only 32 geometric primitives, but most of them (especially the ones in the body of the teapot) are very large, and cover a great number of buckets. This causes the traffic between the master and the slaves to increase. In the third example the object teapot is replicated in order to fit all the space of the scene.

Graphs in Fig. 6 show the obtained speedup factors for the three scenes, rendered at a resolution of 3200×2400 pixels, with buckets of 256×256 pixels each. The tests were run on a cluster of seven PCs running Linux and PVM 3.4.4, each equipped with a 1400MHz Athlon CPU, 512MB of RAM, and a Gigabit Ethernet network adapter. A Gigabit Ethernet switch was used to connect the cluster nodes.

As expected, the frog test leads to a slightly better speedup than the teapot test. Moreover, many teapots provide better speed up factors than a single teapot as slaves are involved in larger computational loads. The overall performance of the system is satisfactory, with a mean speedup around 0.7 to 0.75. The image reconstruction and the bucket assignment parts are intrinsically sequential and cannot be parallelized and these tasks affect the overall performance.

Some extra efficiency can be gained allowing some slaves to work on “non-ready” buckets. With little changes, this would allow to pass data directly between slaves, relieving the master from some of its work. Another improvement for the algorithm could be the use of MPI architecture with the non-blocking receive calls.

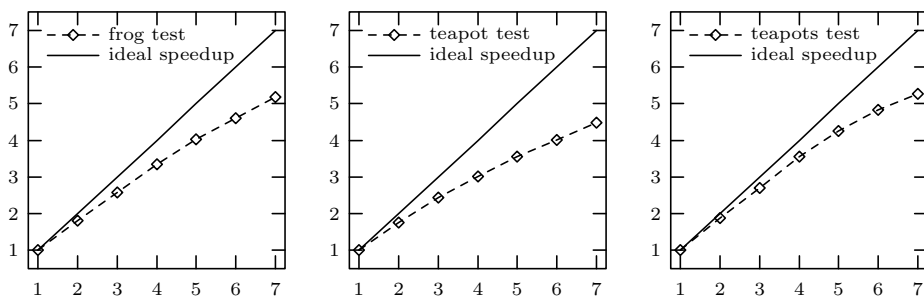


Fig. 6. Speedup factors with different number of slaves for the three test scenes shown in Fig. 5

6 Conclusion and Acknowledgements

This paper proposes an effective distributed implementation of the REYES architecture providing an attractive alternative to other parallel rendering algorithms such as ray tracing and radiosity. The system implements a hybrid solution able both to dynamically distribute the computational load and to avoid unnecessary database duplications.

This project is supported by the Ministero dell'Istruzione dell'Università e della Ricerca in the frame of the Cofin 2001 project: elaborazione ad alte prestazioni per applicazioni con requisiti di elevata intensità computazionale e vincoli di tempo reale.

References

- [1] 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. MIT Press (1994). 165
- [2] Cook, R.L., Carpenter, L., and Catmull, E.: The REYES image architecture. Proc. Siggraph, ACM Comput. Graphics Proceedings, **21** No. 4 (1987) 95–102. 165, 167
- [3] Lin, T.T.Y., and Slater, M.: Stochastic Ray Tracing Using SIMD Processor Arrays. The Visual Computer, **7** No. 4 (1991) 187–199. 166
- [4] Plunkett, D.J., and Bailey, M.J.: The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed. IEEE CG&As, **5** No. 8 (1985) 52–60. 166
- [5] Dippé, M., and Swensen, J.: An Adaptive Algorithm and Parallel Architecture for Realistic Image Synthesis. Proc. Siggraph, ACM Comput. Graphics, **18** No. 3 (1984) 149–158. 166
- [6] Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T., and Shigei, Y.: Load Balancing Strategies for a Parallel Ray-Tracing System based on Constant Sub-division. The Visual Computer, **4** No. 4 (1988) 197–209. 166
- [7] Priol, T., and Bouatouch, K.: Static Load Balancing for Parallel Ray Tracing on a MIMD Hypercube. The Visual Computer, **5** No. 1-2 (1989) 109–119. 166

- [8] Green, S. A., and Paddon, D. J.: Exploiting Coherence for Multiprocessor Ray Tracing. *IEEE CG&As*, **9** No. 6 (1989) 12–26. 166
- [9] Reinhard, E., and Jansen, F. W.: Rendering Large Scenes Using Parallel Ray Tracing. *Parallel Computing*, **23** No. 7 (1997) 873–885. 166
- [10] Scherson, I. D. and Caspary, E.: Multiprocessing for Ray Tracing: a Hierarchical Self-balancing Approach. *The Visual Computer*, **4** No. 4 (1988) 188–196. 166
- [11] Yoon, H. J., Eun, S., and Cho, J. W.: Image parallel Ray Tracing Using Static Load Balancing and Data Prefetching. *Parallel Computing*, **23** No. 7 (1997) 861–872. 166
- [12] Sanna, A., Montuschi, P., and Rossi, M.: A Flexible Algorithm for Multiprocessor Ray Tracing. *The Computer Journal*, **41** No. 7 (1998) 503–516. 166
- [13] Dilger, A.: PVM Patch for POV-Ray.
<http://www-mddsp.enel.ualgary.ca/People/adilger/povray/pvmpov.html>
166
- [14] Plachetka, T.: POV||Ray: Persistence of Vision Parallel Raytracer. *Proc. of SCCG'98*, (1998). 166
- [15] POV-Ray, <http://www.povray.org/> 166
- [16] Recker, R. J., George, D. W., and Greenberg, D. P.: Acceleration Techniques for Progressive Refinement Radiosity. *Computer Graphics*, **24**, No. 2 Proceedings of the 1990 Symposium on Interactive 3D Graphics, (1990) 59–66. 167
- [17] Feda, M., and Purgathofer, W.: Progressive Refinement Radiosity on a Transputer Network. *Photorealistic Rendering in Computer Graphics: Proceedings of the 2nd Eurographics Workshop on Rendering*, (1991), 139–148. 167
- [18] Chalmers, A. G., and Paddon, D. J.: Parallel Processing of Progressive Refinement Radiosity Methods. *Photorealistic Rendering in Computer Graphics: Proceedings of the 2nd Eurographics Workshop on Rendering*, (1991), 149–159. 167

Enhanced File Interoperability with Parallel MPI File-I/O in Image Processing

Douglas Antony Louis Piriya Kumar¹, Paul Levi¹, and Rolf Rabenseifner²

¹ IPVR, Department of Computer Science, University of Stuttgart, Germany
piriyaku@informatik.uni-stuttgart.de

² HLRS, High Performance Computing Center, University of Stuttgart, Germany
rabenseifner@hlrs.de
www.hlrs.de/people/rabenseifner/

Abstract. One of the crucial problems in image processing is *Image Matching*, i.e., to match two images, or in our case, to match a model with the given image. This problem being highly computation intensive, parallel processing is essential to obtain the solutions in time due to real world constraints. The Hausdorff method is used to locate human beings in images by matching the image with models and is parallelized with MPI. The images are usually stored in files with different formats. As most of the formats can be converted into ASCII file format containing integers, we have implemented 3 strategies namely, *Normal File Reading*, *Off-line Conversion* and *Run-time Conversion* for free format integer file reading and writing. The parallelization strategy is optimized so that I/O overheads are minimal. The relative performances with multiple processors are tabulated for all the cases and discussed. The results obtained demonstrate the efficiency of our strategies and the implementations will enhance the file interoperability which will be useful for image processing community to use parallel systems to meet the real time constraints.

Keywords: Supercomputer, Computational Models, File Interoperability, Parallel Algorithms, Image processing and Human recognition.

1 Introduction

The challenging image processing problems in the wide spectrum of defense operations to industrial applications demand run-time solutions which need enormous computing power aptly provided by supercomputers [1]. In most of the core applications problems in Robotics, Satellite Imagery and Medical Imaging, recognizing the crucial parts or items or structures in the given images continues to attract more attention. Thus, the major factor in these computer vision related fields is matching objects in images [2]. Currently, lot of interests are evinced on human motion based on model based approach, and temporal templates [3] with real-time constraints also [4]. A survey of the visual analysis of human movement is presented in [5].

In this paper, an efficient parallel algorithm is developed using the Hausdorff method to ascertain the presence of a human being in the image (or image sequence) by matching the images and the models. In any parallel system, the proper choice of the parallel computing model is the paramount factor. MPI (message passing interface) is used here for its efficiency, portability and functionality [6]. However, due to the domain specific nature of the problem, the images usually stored in files, differ in formats considerably. This poses an impediment to the efficient implementation of the parallel algorithm despite parallel I/O implementations in MPI-2 [7]. As most of the file formats can be converted into ASCII file format in many systems, here we have implemented 3 strategies including one for free format integer file reading and writing. The algorithm is implemented on the supercomputer CRAY T3E with MPI model. A different parallelization method is formulated so that I/O timings are optimized. The time taken for I/O in all the cases are compared with analysis.

The following sections are organized as mentioned here. Section 2 introduces the real application of an image processing problem. The parallel algorithm for image matching is sketched in section 3. In section 4, the File interoperability in MPI is portrayed. The experimental results are analyzed in section 5. Section 6 concludes the paper.

2 Image Processing with Hausdorff Method

The Hausdorff distance [8] is defined as follows: Let the two given finite point sets be $A = a_1, a_2, \dots, a_n$ and $B = b_1, b_2, \dots, b_m$. Hausdorff Distance $H(A, B) = \max(h(A, B), h(B, A))$ where $h(A, B) = \max_{a \in A} \min_{b \in B} EN(a - b)$ where EN is the Euclidean norm (just the distance between the two points).

Here, we have images which have to be analyzed and the models which depict the pattern to be recognized, in our case human beings. Using the general corner detecting algorithm (here SUSAN filter is used [9]), the corners of the images are found which serve as the point set B . Similarly, all the model images are filtered with the same SUSAN filter to find the corners and are stored as point set A . For this application, it is sufficient to find $h(A, B)$. The main Hausdorff distance $H(A, B)$ being a metric has to satisfy the symmetry property and that is why it is defined as maximum of the directed $h(A, B)$ and $h(B, A)$. The model is placed over all possible positions on the image and for each such position, $h(A, B)$ is computed which is computational intensive. The model is found to be present in the image if $h(\text{model}, \text{image})$ is less than some predefined threshold. Against various models, the image is matched using the method.

3 Parallel Algorithm for Image Matching

3.1 Parallelization

For this particular problem, there can be at least three methods to parallelize. One way is to take each image by a processor and match with all models. The

other way is to take one by one all images by all processors and divide the set of models equally among the processors. The third way is to divide each time one image by the number of processors and match that portion of image with all models. In the last model, overlapping is essential to get the proper solution. The second model is preferable in the situation while tracking a person is the major factor. In this paper, the second method is implemented. A good insight to various strategies of parallelization can be found in [10], [11] and [12].

3.2 Outline of Parallel Program

Let r be the number of images, q be the number of models and p be the number of processors.

```

MPI_Comm_rank(MPI_COMM_WORLD,&j);
MPI_Comm_size(MPI_COMM_WORLD,&p);
for each image i=1..r do
{ MPI_File_open(...,MPI_COMM_WORLD,...);
  MPI_File_read(...); /*all processors read  $i^{th}$  image */
  MPI_File_close(...);
  for each model k = j, j+p, j+2*p, .. q do
  { MPI_File_open(...,MPI_COMM_SELF,...);
    MPI_File_read(...); /*each processor j reads only the corresponding model k */
    MPI_File_close(...);
    h = Hausdorff distance of the image i matched with model k for all positions.
    h_partial_min = min ( h, h_partial_min);
    whenever the h(k,i) < threshold, this position is notified.
    /* if required the best matching model is also computed depending upon
    the minimum threshold. */
  }
  MPI_Allreduce(h_partial_min,&h_min,1,MPI_FLOAT,MPI_MIN,MPI_COMM_WORLD);
}

```

4 Parallel I/O and File Interoperability in MPI-2

As the vital focus of this paper is not on parallel processing, the I/O operations especially file related operations are investigated. The significant optimizations required for efficiency can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files [7]. Parallel reading of the same image or model into the memory of several MPI processes can be implemented with the `MPI_File_read_all`. This collective routine enables the MPI library to optimize reading and broadcasting the file information into the memories of all processes. In image processing, there exists also a huge number of different formats to store the image data in files. The standard image processing software gives the options of a proprietary format or a standard ASCII format. Because most of the formats can be converted into ASCII file format in many systems, and to circumvent problems with the 64-bit internal integer format on the Cray T3E, we have chosen to use only an ASCII format. Therefore, it is mandatory to implement the conversion of ASCII file (mostly representing integers being pixel coordinates and gray values) so that file interoperability in MPI can be used

effectively for image processing. As the sizes of the files increase obviously the I/O overheads also increase. In image processing, there will be always many files both for images and models. Hence, it is not only the sizes of the images, but also the number of them is a matter of concern for I/O overheads.

The file interoperability means to read and write the information previously written or read respectively to a file not just as bits of data, but the actual information the bits represent. File interoperability has three aspects namely, 1. transferring the bits, 2. converting different file structures and 3. converting between different machine representations. The third being the concern here, the multiple data representations and the inability to read integer data stored in an ASCII file which is needed for image processing are explained in the following subsection.

4.1 Data Representations

MPI-2 defines the following three data representation, 1. *native*, 2. *internal* and 3. *external32* [7]. In *native* representation, the data is stored in a file exactly as it is in memory. In *external32* format, also a binary data representation is used. Obviously, it is impossible to use these formats directly to read integer data from ASCII files. The *internal* representation cannot be used for data exchange between MPI programs and other non-MPI programs that have provided the image data, because the internal representation may be chosen arbitrarily by the implementer of the MPI library. MPI-2 has standardized also a method to use *user-defined* data representation. Here, the user can combine the parallel I/O capabilities with own byte-to-data conversion routines. The major constraint is that the representation of a given data type must have a well-defined number of bytes. As the number of digits of integers in an ASCII file vary (and each integer may end either with a blank or an end-of-line character), user-defined representation also cannot help reading integers efficiently from ASCII files.

4.2 Reading Integer Data from ASCII File with MPI I/O

The former constraints force the implementation of the following strategies:

Normal File Reading with *fscanf* In this first strategy, the files are read using normal file reading command *fscanf* instead of MPI for the sake of comparison with MPI file I/O operations. It may be recalled that there is no need for conversion as *fscanf* can directly read the integers from the files.

Off-line Conversion In this second strategy, the ASCII file is converted into a native file by a separate program. This gives the facility to convert the required ASCII file off-line which enables the image processing program to read the native file without any difficulty. To achieve heterogeneity, MPI external 32 data representation can be used instead of the native format.

Runtime Conversion In this third strategy, the entire ASCII file is read into a large buffer of type CHAR, and then individually by reading every character till it is terminated either by a blank or by an end-of-line character, the same is

Table 1. Parallelization scheme of I/O and computation

| ranks=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Remarks |
|----------|----------|----------|----------|----------|----------|----------|----------|---|
| R(0,0) | R(1,1) | R(2,2) | R(3,3) | R(0,4) | R(1,5) | R(2,6) | R(3,7) | read first images&models |
| c(0,0) | c(1,1) | c(2,2) | c(3,3) | c(0,4) | c(1,5) | c(2,6) | c(3,7) | exchange models |
| r(1) | r(2) | r(3) | r(4) | r(5) | r(6) | r(7) | r(0) | |
| c(0,1) | c(1,2) | c(2,3) | c(3,4) | c(0,5) | c(1,6) | c(2,7) | c(3,0) | |
| r(2) | r(3) | r(4) | r(5) | r(6) | r(7) | r(0) | r(1) | |
| c(0,2) | c(1,3) | c(2,4) | c(3,5) | c(0,6) | c(1,7) | c(2,0) | c(3,1) | read next models |
| r(3) | r(4) | r(5) | r(6) | r(7) | r(0) | r(1) | r(2) | |
| c(0,3) | c(1,4) | c(2,5) | c(3,6) | c(0,7) | c(1,0) | c(2,1) | c(3,2) | |
| R(8) | R(9) | R(10) | R(11) | R(12) | R(13) | R(14) | R(15) | |
| c(0,8) | c(1,9) | c(2,10) | c(3,11) | c(0,12) | c(1,13) | c(2,14) | c(3,15) | read next image & models (with reverse sequence, q = # of models) |
| r(9) | r(10) | r(11) | r(12) | r(13) | r(14) | r(15) | r(8) | |
| c(0,9) | c(1,10) | c(2,11) | c(3,12) | c(0,13) | c(1,14) | c(2,15) | c(3,16) | |
| ... | ... | ... | ... | ... | ... | ... | ... | |
| R(4,q-1) | R(5,q-2) | R(6,q-3) | R(7,q-4) | R(4,q-5) | R(5,q-6) | R(6,q-7) | R(7,q-8) | |
| ... | ... | ... | ... | ... | ... | ... | ... | |

converted into an integer at run-time. In fact, the original file remains as ASCII file and is still used. The conversion can be stored as a native file for further use, if need be. It may be recalled the ASCII to Integer conversion function is very easy to implement which is also system independent.

4.3 Optimizing the Parallel I/O

The image data usage pattern has two chances for optimization: (a) all image data must be reused (and probably reloaded) for comparing with several models, and (b) all models must be reused (and probably reloaded) for comparing with several images. In the sequential version of the software, each image is loaded once and all models are loaded again for comparing with each image. By reversing the sequence of models for each even image number, at least the latest models can be cached in memory. In the first parallel version loading of the images can be optimized with collective reading into all processes.

If more than one image can be analyzed in parallel, i.e., if one can accept an additional delay for the analysis of an image because not all available processors are used for analyzing and because the start of the analysis is delayed until a set of images is available, then the parallelization can be optimized according to the scheme in Table 1. The scheme shows the analysis of 4 images in parallel on 8 processors. R(i,k) denotes reading of the image i and model k, R(k) is only reading of model k, r(k) is receiving of model k with point-to-point communication from the right neighbor (sending is omitted in the figure), and c(i,k) denotes the computation of the Hausdorff distance for image i and model k.

Looking at the scheme, note that reading the image into several processors at the same time (e.g., image 0 into processes 0 and 4) can be still optimized with collective reading (MPI_File_read_all) that internally should optimize this operation by reading once from disk and broadcasting the image data to the processes.

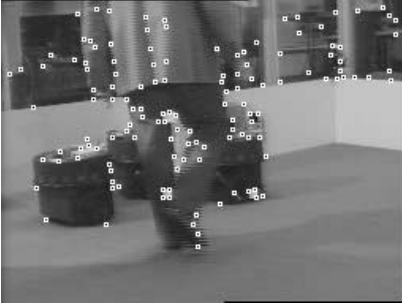


Fig. 1. A sample Image



Fig. 2. A sample Model

Reading several images and models at the same time can be accelerated by the use of striped file-systems. The scheme is also optimized for a cluster of shared memory nodes (SMPs). The vertical bar between rank 3 and 4 may denote such a boundary between SMPs. One can see on each node, that only one model is received from another node (and another model is sent) while exchanging all models.

5 Results and Analysis

For the purpose of illustration, four sample images (one shown in Fig. 1) and four models (one shown in Fig. 2) are considered. The algorithm is tested with 1, 2 and 4 processors on the Cray T3E-900 at HLRS. As the interest of the paper is on I/O, the I/O timings per process are tabulated in Table 2 for 4 images and 4 models. The timing is done with `MPI_Wtime()`. Before starting each I/O timing, a barrier is done to prohibit that any synchronization time is assessed as I/O time. Although the I/O requires only a small part of the total execution time in the current version of the program, it is expected that on faster processing systems and with better optimization of the Hausdorff algorithm, I/O will be a relevant factor of execution speed. In the *original* parallelization, each image is read by all processes (which may be optimized by the MPI library), and for each image, each process reads only a subset of the models according to the numbers of processors. In the *optimized* parallelization, each image is read by only one process, and for each set of images analyzed in parallel, each model is read only once and then transferred with message passing to the other processes. Table 3 shows the accumulated number of reading an image or model file or transferring a model for our test case with 4 images and 4 models.

We started our experiments with normal reading with `fscanf`. Our original parallelization resulted in a larger I/O time because each image had to be read on each processor again. In the second experiment we parallelized this reading and substituted each `fscanf` by MPI-2 file reading. Because reading of ASCII integers is not available in MPI-2, we have chosen reading characters. Normally

Table 2. I/O time per process: wall clock time per process to handle the reading of 4 images and 4 models, including repeated reading or message exchanges of the model.

| No. | Parallelization | File Op | Conversion | I/O Entities | 1 proc | 2 proc | 4 proc |
|-----|-----------------|---------|------------|----------------|---------|---------|---------|
| 1 | Original | fscanf | On-line | integers | 0.126 s | 0.130 s | 0.142 s |
| 2 | Original | MPI | On-line | characters | 7.087 s | 6.173 s | 6.563 s |
| 3 | Original | MPI | On-line | whole file | 0.157 s | 0.196 s | 0.234 s |
| 4 | Original | MPI | Off-line | 3*int, 2*array | 0.189 s | 0.182 s | 0.195 s |
| 5 | Optimized | MPI | On-line | whole file | 0.163 s | 0.071 s | 0.040 s |
| 6 | Optimized | fscanf | On-line | integers | 0.129 s | 0.068 s | 0.036 s |

Table 3. Each entry shows the accumulated number of *images read + models read + models exchanged* by all processes with the different parallelization schemes, e.g., 4*2+16+0 means, that 4 times 2 identical images, and 16 models are read, and 0 models are exchanged by message transfer

| Parallelization | accumulated number of images + models read with | | |
|-----------------|---|--------------|--------------|
| | 1 process | 2 processes | 4 processes |
| Original | 4*1 + 16 + 0 | 4*2 + 16 + 0 | 4*4 + 16 + 0 |
| Optimized | 4*1 + 16 + 0 | 4*1 + 8 + 8 | 4*1 + 4 + 12 |

each integer is expressed only with a few characters, therefore, the expected additional overhead was not expected very high. But the measurements have shown that this solution was 46 times slower than the original code. The MPI-2 I/O library on the Cray T3E could not be used in a similar way as `fscanf()` or the `getc()` can be used. To overcome the high latency of the MPI I/O routines, reading the whole file with one (experiment No.3) or only a few (No.4) MPI operations was implemented. But there is still no benefit from parallelizing the I/O. The I/O time per process grows with the number of processes and the accumulated I/O time with 4 processors is therefore 4–6 times more than with one processor. In the last two experiments, the parallelization was optimized to reduce the number of reading of each image and model. This method achieves an optimal speedup for the I/O. But also with this optimization, the `fscanf` solution is about 10% faster than the MPI I/O solution on 4 processes.

These experiments have shown that (a) MPI I/O can be used for ASCII files, (b) but only large chunks should be accessed due to large latencies of MPI I/O routines, and (c) optimizations that can be implemented by the applications should be preferred of optimizations that may be done inside the MPI library, (d) as long as many small or medium ASCII files should be accessed, it may be better to use standard I/O by many processes and classical message passing or broadcasting the information to all processes that need the same information, than using collective MPI I/O.

6 Conclusion

One of the computationally intensive image processing problem, *Image matching* which demands the solutions within real time constraints is investigated focusing the attention on MPI File Interoperability especially with ASCII files. Due to the domain specific nature of the problem, the images usually stored in files, differ in formats considerably. This poses an impediment to the efficient implementation of the parallel algorithm despite parallel I/O implementations in MPI-2. As most of the formats can be converted into ASCII file format in many systems, the three strategies namely, *Normal File Reading*, *Off-line Conversion* and *Run-time Conversion* for free format integer file reading and writing are implemented on Cray T3E with MPI-2. The modified parallelization presented in the paper produced better results comparing the I/O timings. The important conclusion of the paper is that the problem of file format conversion in image processing applications can be efficiently solved with the proper parallelization and MPI parallel I/O operations. In all the images, the accurate positions (to one pixel resolution) of the human beings with the corresponding best model are not only found correctly but also efficiently as the obtained results demonstrate.

References

- [1] Ian Foster and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1998. 174
- [2] Douglas A.L.Piriyakumar, and Paul Levi, "On the symmetries of regular repeated objects using graph theory based novel isomorphism", *The 5th International Conference on PATTERN RECOGNITION and IMAGE ANALYSIS*, 16 - 22 October, 2000, Samara, The Russian Federation. 174
- [3] Aaron F.Bobick and James W.Davis., "The Recognition of Human Movement using Temporal Templates", *IEEE Trans. PAMI*, vol.23, no.3, pp.257-267, March, 2001. 174
- [4] N. T. Siebel and S. J. Maybank, "Real-time tracking of Pedestrians and vehicles", *IEEE International workshop PETS'2001*. 174
- [5] D. M. Gavrilu, "The Visual Analysis of Human Movement: A Survey", *Computer Vision and Image Processing*, vol. 73, no. 1, pp. 82-98, 1999. 174
- [6] William Gropp, Ewing Lusk and Anthony Skejellum., *Using MPI*, MIT press, 1995. 175
- [7] MPI-2, Special Issue, *The International Journal of High Performance Computing Applications*, vol. 12, no. 1/2, 1998. 175, 176, 177
- [8] Daniel Huttenlocher, Gregory Klanderman and William Rucklidge., "Comparing images using Hausdorff distance", *Transaction on PAMI*, vol. 15, no. 9, pp. 850-863, September, 1993. 175
- [9] S. Smith and J. Brady, "SUSAN - a new approach to low level image processing", *Int. Journal of Computer Vision*, vol. 23, no. 1, pp. 45-78, 1997. 175
- [10] Armin Baeumker and Wolfgang Dittrich., "Parallel algorithms for Image processing: Practical Algorithms with experiments", Technical report, Department of Mathematics and Computer Science, University of Paderborn, Germany, 1996. 176

- [11] J. F. JaJa, Introduction to Parallel Algorithms. Addison-Wesley, 1992. 176
- [12] Rolf Rabenseifner, Parallel Programming Workshop Course Material, Internal report 166, Computer Center, University of Stuttgart, 2001.
http://www.hlrs.de/organization/par/par_prog_ws/ 176

Granularity Levels in Parallel Block-Matching Motion Compensation

Florian Tischler and Andreas Uhl*

Salzburg University, Department of Scientific Computing
Jakob Haringer-Str. 2, A-5020 Salzburg, Austria
{ftisch,uhl}@cosy.sbg.ac.at
<http://www.cosy.sbg.ac.at/sc/>

Abstract. We discuss different granularity levels for parallel block-based motion compensation as used in video compression. The performance of the corresponding MPI implementations on a SGI Power Challenge, a Cray T3-E, and a Siemens HPCline cluster is compared.

1 Introduction

The widespread use of digital video in various environments has caused a high demand for efficient compression techniques. Unfortunately, many compression algorithms have prohibitive execution times using a single serial microprocessor [7], which leads to the use of high performance computing systems for such tasks. Software based approaches are becoming more popular in this area because of the rapid evolution of multimedia techniques which has dramatically shortened the time available to come up with a new hardware design for each improved standard. In this context, several papers have been published describing real-time video coding on general purpose parallel architectures – see for example MPEG-1,2,4 [1, 5, 8], H.261/3 [6], and wavelet-based video compression [3].

Block-matching motion compensation is the most demanding part of current video coding standards and requires 60 – 80 % of the total computations involved. A significant amount of work discusses dedicated hardware for block-matching (e.g. [2]), some software based block-matching approaches have also been investigated [9]. However, in most cases, only one approach (in particular one parallelization granularity), is discussed in detail and it is left to the reader to compare the results to other research in this area. In this work, we systematically compare different levels of parallelization granularity from the scalability point of view and consider as well requirements coming from the applications' side.

In section 2 we shortly review block-matching motion compensation in the video coding context. Section 3 discusses four granularity levels for parallel block-matching which are experimentally evaluated on three architectures using MPI based message passing in section 4.

* The authors have been partially supported by the Austrian Science Fund FWF, project no. P13903.

2 Block-Matching Motion Compensation in Video Coding

The main idea of motion compensated video coding is to use the temporal and spatial correlation between frames in a video sequence for predicting the current frame from previously (de)coded ones. Since this prediction fails in some regions (e.g., due to occlusion), the residual between this prediction and the current frame being processed is computed and additionally stored after lossy compression.

Because of its simplicity and effectiveness block-matching algorithms are widely used to remove temporal correlation [4]. In block-matching motion compensation, the scene (i.e. video frame) is classically divided into non-overlapping “block” regions. For estimating the motion, each block in the current frame is compared against the blocks in the search area in the reference frame (i.e. previously encoded and decoded frame) and the motion vector (d_1, d_2) corresponding to the best match is returned (see Fig. 1). The “best” match of the blocks is identified to be that match giving the minimum mean square error (MSE) of all blocks in search area defined as

$$MSE(d_1, d_2) = \frac{1}{N_1 N_2} \sum_{(n_1, n_2) \in \mathcal{B}} [s_k(n_1, n_2) - \hat{s}_{k-l}(n_1 + d_1, n_2 + d_2)]^2$$

where \mathcal{B} denotes a $N_1 * N_2$ block for a set of candidate motion vectors (d_1, d_2) , s is the current frame and \hat{s} the reference frame.

The algorithm which visits all blocks in the search area to compute the minimum is called full search (FS). In order to speed up the search process, many techniques have been proposed to reduce the number of candidate blocks. The main idea is to introduce a specific search pattern which is recursively applied at the position of the minimal local error. The most popular algorithm of this type is called “Three Step Search” (TSS) which reduces the computational amount significantly at the cost of a suboptimal solution (and therefore a residual with slightly more energy).

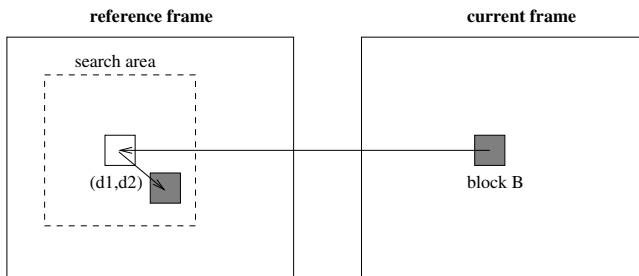


Fig. 1. Block-Matching motion estimation

3 Parallel Block-matching Algorithms with Different Granularity

In general, granularity determines important characteristics of a parallelization approach. A coarse grained parallelization usually requires little communication, on the other hand, balanced load may be hard to achieve. Independent of the results corresponding to parallelization efficiency, granularity has also major impact with respect to hardware requirements and coding delay of a video compression system (compare also Table 1 at the end of section 3).

Examining the sequence of frames in the video stream to be encoded, we identify two main granularity levels: intra-frame granularity (fine grained parallelization) and inter-frame granularity (coarse grained parallelization).

3.1 Intra-frame Granularity

As described in the previous section, the current frame and the reference frame are segmented into equal sized blocks. In the *Blockbased* parallelization approach, the blocks of the current frame are distributed among the processing elements (PE), the computation of the motion vector for each block is done locally. To avoid interprocess communication, the entire search window surrounding the corresponding block in the reference frame (see Fig. 2.a) is sent to the PE in charge of the current block resulting in an overlapping data partition. Given the size of the maximal allowed motion vector (s_x, s_y) and blocks with size $(b_x * b_y)$, the amount for the overlapping parts to be sent is maximal $2s_x b_y + 2s_y b_x + 4s_x s_y$ per block (the light gray shaded areas in Fig. 2.a show the actual area sent for the local computation).

Another way of partitioning the current frame among the PE is the stripe subimage method (see [9] for a comparison of several flavours of this method) which we denote *Intra Frame* parallelization. Using this method, the current

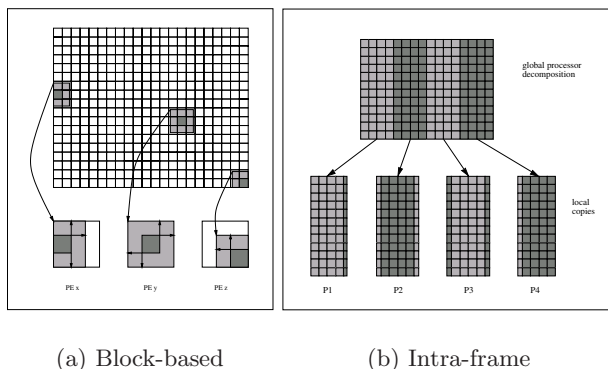


Fig. 2. Partitioning schemes: fine granularity

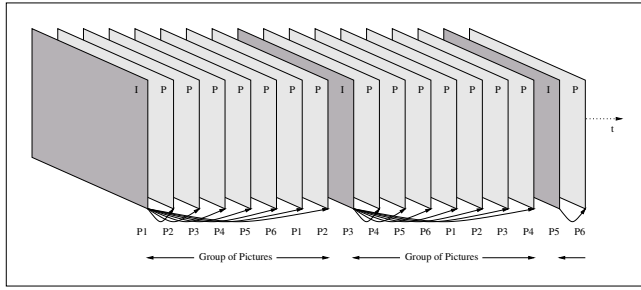


Fig. 3. Partitioning with coarse granularity

frame is split into horizontal or vertical stripes as shown in Fig. 2.b and the computations associated with the blocks contained in one contiguous stripe are assigned to one PE. Again, a search-overlap is used to avoid interprocess communication among the PEs. Given the size of the maximal allowed motion vector (s_x, s_y) and frames of the size $(f_x * f_y)$, the amount for the overlapping parts to be sent is maximal $f_x * s_y * k$ per stripe where $k = 1$ for stripes located at the edge of the frame and $k = 2$ for “inner” stripes (for partitioning in vertical direction).

To preserve the quality reached by the sequential version, the smallest allowed unit for splitting the frame is limited by the blocksize. These units are combined according to the number of PEs to form stripes as uniformly sized as possible. Therefore, this method should perform well especially in the case where the units can be distributed evenly among the workers, i.e. when $\frac{f_y}{b_y}$ is an integer multiple of the number of PEs in the case of vertical splitting.

In both fine grained parallelization approaches, data is distributed among the PEs in round robin fashion. Results are collected after completing the computations associated with one block/stripe (blocking MPI communication) or frame (non-blocking MPI communication).

3.2 Inter-frame Granularity

A group of pictures (GOP) is a collection of P-frames (predicted frames) which are all processed in relation to one reference frame, the I-frame. A more coarse grained way to distribute the block-matching computations is to assign entire frames to single PEs. Fig. 3 illustrates this technique which is denoted GOP parallelization.

A second possibility is to assign complete GOPs to single PEs (denoted eGOP). Note that this approach is not investigated further experimentally since coding delay and memory requirements are too high for practical systems. Moreover, loadinbalance is significant for short videos using this technique.

In Table 1, we compare all four granularity levels with respect to their memory requirement per PE (MR), the minimal and maximal coding delay (CD –

Table 1. Properties of the parallelization approaches

| | Block Based | Intra Frame | GOP | eGOP |
|--------------|---------------|------------------|----------------|--------------|
| MR current | Block | Stripe | Frame | GOP |
| MR reference | Search window | Stripe + overlap | Frame | – |
| min. CD | – | – | – | 1 GOP |
| max. CD | – | – | $p - 1$ frames | $p - 1$ GOPs |
| comm. # | 2 per block | 2 per stripe | 2 per frame | 2 per GOP |

the lowest value is considered to be 1 frame), and the number of communication events involved. The number of PEs = p .

We may derive immediately that coarse grained parallelization (GOP and eGOP) will not be suited for real-time and on-line applications like video conferencing, since especially the high coding delay is not acceptable for these types of applications. Additionally, the high memory requirements lead to high costs (especially for hardware solutions) and poor cache performance.

4 Experiments

4.1 Experimental Settings

All simulations use the standard test sequence “Football” in QCIF (quarter common intermediate format), having a size of 176×144 pixels. The blocksize is set to 8×8 , motion vectors may point 7 pixels in each direction, the computations are performed on 40 successive frames of the sequence. Three different architectures using native MPI versions are employed: a SGI Power Challenge (20 MIPS R10000 processors and 2.5 GB memory), a Cray T3E (DEC Alpha EV5 processors, 128 MB memory each, interconnected by a 3D torus with 3 GB/s bandwidth in each network node), and a Siemens HPCline cluster (compute nodes with 2 Pentium III @ 850 MHz and 512 MB memory each, interconnected by a 2D torus with 500 MB/s bandwidth in each network node). Note that the three systems have opposite properties. Whereas the SGI has the slowest compute nodes and the cluster the fastest, the communication system is fastest on the SGI (shared memory) and slowest on the cluster (see above). Therefore, we may expect the best result with respect to scalability on the SGI and the worst results on the cluster, the Cray is expected to perform in-between. SPMD-style programming with non-blocking MPI communication commands is used.

Fig. 4.a compares speedup of Block-based parallelization on the SGI employing FS and TSS with classical precision (denoted -s1) and quarter-pixel accuracy (-s4) as used in recent standards. As it is expected, speedup improves with increasing complexity since the communication amount is constant for all four settings. In the following, FS-s1 is used. Fig. 4.b compares two different communication patterns: the classical host–node scheme (FS) and a scheme employing an additional result node (FS-S). Interestingly, in the second approach the additional communication node compensates the missing compute node and this

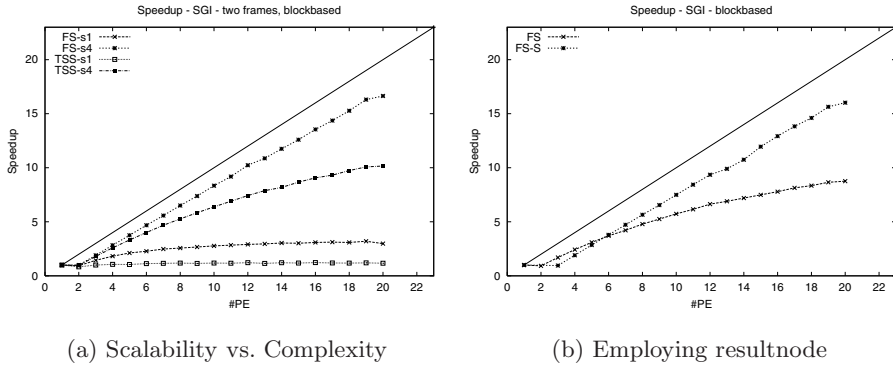


Fig. 4. Speedup SGI Power Challenge: influence of computational demand and communication pattern (Block-based mode)

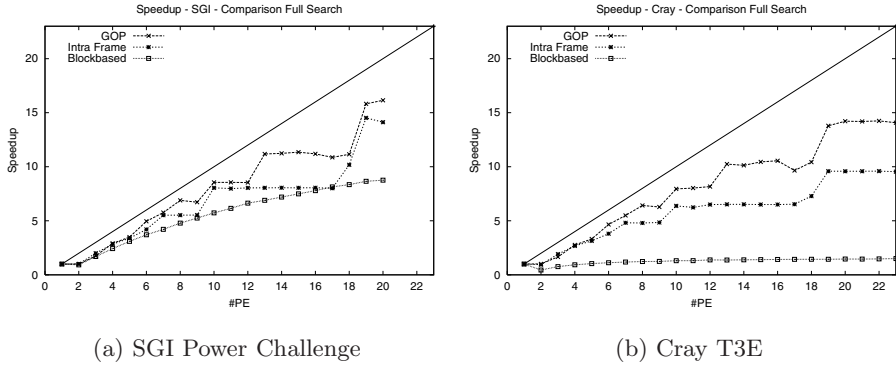
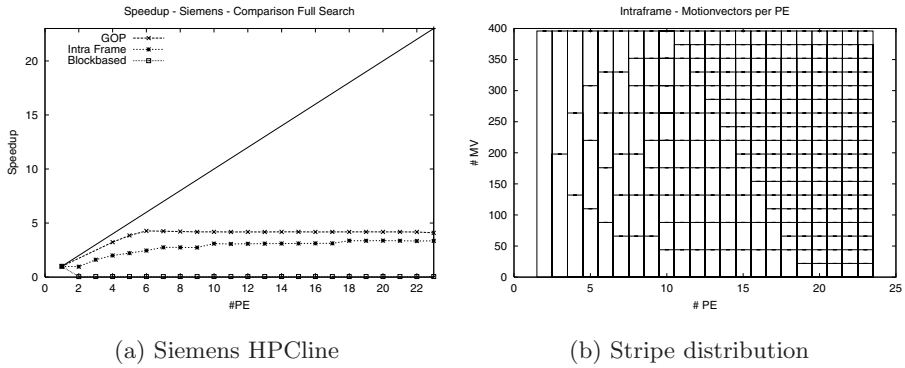
results in better scalability. However, this is true only for this communication intense setting (Block-based parallelization), therefore we restrict the following results to the classical approach.

4.2 Experimental Results

Block-based parallelization shows satisfactory speedup on the SGI only (see Fig. 5.a – but even here it shows the poorest performance of all three granularities). Obviously, the communication demand is too high to deliver reasonable results, even though the load may be balanced perfectly. Intra Frame parallelization exhibits several plateaus in the speedup plots. To explain this phenomenon, see Fig. 5.b: The first plateau can be found from seven PEs up to nine PEs, the second is in the range from ten to eighteen PEs, and the last is from nineteen PEs onwards. Since we split the frames along the vertical axis, we can have at most eighteen stripes with blocksize eight. This fact explains the last plateau.

The remaining plateaus are due to the unequal size of stripes in this scheme. In Fig. 6.b, each bar depicts the number of displacement vectors, that have to be computed by each PE. The running time of the parallel execution is determined by the largest bar in each column – as long as the number of units (see section 3.1) is an integer multiple of the number of working PEs, we result in balanced load (equal sized bars). For a small number of PEs, additional PEs facilitate a reduction of the size of the largest bar (but unbalanced load, e.g. 4 and 5 working PEs). For a larger number of PEs, additional PEs do not necessarily reduce the size of the largest bar for a certain number of PEs, which leads to the plateaus.

Also, GOP parallelization leads to plateaus in the speedup plots. However, the overall performance is the best compared to the other two approaches. The reason for these plateaus is that a fixed number of frames can not be distributed

**Fig. 5.** Speedup Results**Fig. 6.** Speedup Results and stripe distribution for Intra Frame parallelization

evenly among an arbitrary number of PEs, therefore several PEs may remain idle when the last frames are processed. Note, that these plateaus disappear for long video sequences since unbalanced load in the last scheduling round is not important for a large number of scheduling rounds. This is not the case for Intra Frame parallelization since these plateaus appear on a per frame basis and can only be avoided by skipping synchronization at the frame level, which is difficult from the application point of view and reduces the corresponding advantages of this approach. When comparing the three architectures, the performance matches exactly the prediction. The Siemens cluster is hardly suited to process QCIF-sized video, whereas reasonable performance is achieved on the Cray and the SGI using GOP and Intra Frame parallelization. Only on the SGI, especially when using an additional data sink for result collection (compare Fig. 4.b), Block-based parallelization performs well.

5 Conclusion and Future Work

We have compared different granularity levels of parallel block-matching motion compensation in the context of video coding. Corresponding to the room for improvement in Block-based parallelization due to communication overhead and of GOP parallelization due to load imbalance at the end of the video, we will investigate two hybrid modes in future work: assigning groups of blocks with a priori computed size to the PEs and GOP parallelization with a switch to finer granularity when the end of the video sequence is reached.

References

- [1] S.M. Akramullah, I. Ahmad, and M.L. Liou. Performance of software-based MPEG-2 video encoder on parallel and distributed systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4):687–695, 1997. 183
- [2] S.-C. Cheng and H.-M. Hang. A comparison of block-matching algorithms mapped to systolic-array implementation. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(5):741–757, October 1997. 183
- [3] M. Feil and A. Uhl. Efficient wavelet-based video coding. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium IPDPS'02 (Abstracts and CD-ROM)*, PDIVM 2002, page 128. IEEE Computer Society Press, 2002. 183
- [4] B. Furht, J. Greenberg, and R. Westwater. *Motion estimation algorithms for video compression*. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands, 1997. 184
- [5] Y. He, I. Ahmad, and M.L. Liou. Modeling and scheduling for MPEG-4 based video encoder using a cluster of workstations. In P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Computation. Proceedings of ACPC'99*, volume 1557 of *Lecture Notes on Computer Science*, pages 306–316. Springer-Verlag, 1999. 183
- [6] K. K. Leung, N. H. C. Yung, and P. Y. S. Cheung. Parallelization methodology for video coding – an implementation on the TMS320C80. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(10):1413–1423, 2000. 183
- [7] K. Shen, G.W. Cook, L.H. Jamieson, and E.J. Delp. An overview of parallel processing approaches to image and video compression. In M. Rabbani, editor, *Image and Video Compression*, volume 2186 of *SPIE Proceedings*, pages 197–208, 1994. 183
- [8] K. Shen, L.A. Rowe, and E.J. Delp. A parallel implementation of an MPEG1 encoder: faster than real-time ! In A. A. Rodriguez, R. J. Safranek, and E. J. Delp, editors, *Digital Video Compression: Algorithms and Technologies*, volume 2419 of *SPIE Proceedings*, pages 407–418, 1995. 183
- [9] M. Tan, J.M. Siegel, and H.J. Siegel. Parallel implementation of block-based motion vector estimation for video compression on four parallel processing systems. *International Journal of Parallel Programming*, 27(3):195–225, 1999. 183, 185

An Analytical Model of Scheduling for Conservative Parallel Simulation

Ha Yoon Song¹, Junghwan Kim², and Kyun Rak Chong¹

¹ College of Information and Computer Engineering, Hongik University, Seoul, Korea
{song, chong}@cs.hongik.ac.kr

² Department of Computer Science, Konkuk University, Chungju, Korea
jhkim@kku.ac.kr

Abstract. The critical path provides a lower bound on the execution time of a distributed discrete event simulation. The optimal execution time can be achieved by immediately executing each event on the critical path. A solution is to preferentially schedule an LP, called a critical parent, which probably has a critical event. Critical parent preferential scheduling has been experienced as a viable solution of conservative LP scheduling. In this paper, we introduce a mathematical analysis to show the quantitative benefit of critical parent preferential scheduling in terms of speed-up of execution time. The analysis shows the benefit of the critical parent preferential scheduling with the acceleration of earliest incoming time update and with the reduction of non-critical null messages. The effectiveness of analytical model has been validated with experiments.

1 Introduction

There are two widely used families of parallel simulation protocols: conservative [1] and optimistic [2]. Conservative techniques only execute safe events, while optimistic algorithms execute any available events, but must roll back to prior states to rectify the incorrect executions of events.

Practically, the scheduling policy used by a processor to schedule simulation objects (commonly called *logical processors* or LPs) can have significant impacts on simulation performance. In general, the execution time of a simulation model is constrained by its critical path (the longest chain of causally dependent events). In certain cases, optimistic algorithms with lazy cancellation and a little luck can beat the critical path [3]. Intuitively, an optimal scheduling algorithm for a conservative simulation will schedule events on the critical path prior to other events. Unfortunately, it is practically impossible to identify dynamically which event or LP might be on the critical path is. In a Chandy-Misra style deadlock avoidance algorithm, so called *null message* algorithm, the awaited input may be the null message that will allow the LP to advance its safe time to the point that it can execute its next event. If many LPs are waiting for the same predecessor, it could be that the predecessor is on the critical path, and scheduling priority can be given to that LP.

Several scheduling algorithms have been proposed for the conservative simulation. Naroska et al. [4] introduced an algorithm of conservative scheduling which gives preference on so called critical LPs which directly influence on the other LPs. Song et al. [5] experimentally compared several heuristics to schedule critical events first and showed that scheduling LPs with the least *Earliest Outgoing Time* (EOT) may cope with various simulation models with good performance. Xiao et al. [6] provided a centralized multi-level scheduling called *Critical Channel Traversing* (CCT). Liu et al. [7] researched a removal of extra locking overhead of CCT scheduling. In this paper, we will provide a quantitative analysis of a *Critical Parent Preferential Scheduling* (CPPS) that preferentially schedules the LPs with the least EOT values based on the conservative protocols specified in [5], compared with the unsorted random scheduling (URS).

The remainder of this paper is organized as follows. Section 2 will describe the CPPS mechanism. We will introduce an analytical approach for CPPS scheduling with null message algorithm in section 3. A detailed study for verification will be shown with experiments in section 4. The final section will summarize the results and remark the future research.

2 Critical Parent Preferential Scheduling

All things being equal, improving the rate of EIT advancement of an LP will tend to reduce execution time of the model. The EIT of an LP is determined by its parent LP with the smallest EOT, and can only be advanced upon receipt of an updated EOT in a form of a null message from that predecessor which is called the *critical parent*. If we assume that the next event at an LP is on the critical path, then the critical parent is the one blocking execution of the critical event. The chains of critical parent from a source LP leading to a certain LP is called a *Precedence Critical Path* (PCP) of the LP. Critical parent and the PCP are dynamically changed during simulation with EIT updates while they remain unchanged between two consequent EIT updates. The details of critical path and critical parent can be found in [8] as well as the details of conservative simulation protocol. The best way to probabilistically identify a critical parent is to select an LP with least EOT value among the LPs mapped on a CPU. An LP with the least EOT value could be a critical parent of one of its successors. Even if it is not a critical parent, i.e. a real critical parent resides on another CPU, it is still true that the LP is the slowest one on the CPU and highly probably to be a critical parent of another LP, therefore, it should be preferentially scheduled. Of course, each LP may have a different critical parent, and only one LP per CPU can be scheduled at a moment. Practically, one of the critical parents on each processor is designated as the *critical LP*, and is scheduled first. The implementation is to sort LPs with their EOT values and schedule the LP with the least EOT value first. Several LPs usually partitioned on a set of CPUs of a parallel or distributed simulation system. Thus it is required to schedule a set of LPs on a CPU. The null message algorithm might work well with the URS that is round-robin style random scheduling for fairness with doubly linked list queue. Each LP in turn

executes all its safe events and all queue operations have $O(1)$ complexity. CPPS schedules the LP with the smallest EOT values first among the LPs mapped on a CPU. To implement CPPS with traditional null message algorithm, the LPs in the scheduling queue must be sorted by EOT values of LPs in an ascending order. Among the multiple varieties [9] of sorting algorithms, the splay tree [10] is incorporated which has $O(\log N)$ insertion time and $O(1)$ removal time.

3 Scheduling Benefit Analysis

In this section, we analyze the benefit of CPPS over URS. The acceleration ratio of EIT update, children LP acceleration in terms of null message reduction, and their combination will be discussed.

3.1 EIT Update Acceleration

The critical parent preferential scheduling usually accelerates the starting of new EIT update. Let $T(eit_i)$ be a physical time of EIT update at LP_i , $T(eot_i)$ be a physical time of EOT determination at LP_i with given EIT, and T_m be the average point-to-point message processing time of inter processor null messages. As well, eit_i is EIT value of LP_i and eot_i is EOT value of LP_i . We assume $T(eot_i) = T(eit_i)$ if LP_i is the first processor to be scheduled on a processor since it only requires relatively smaller number of processor cycles. EOT of an LP can be computed using EIT and the local status of the LP when the LP is scheduled to run on a CPU. Let PP_a be a CPU named a . For the LP_i on PP_a , we denote it as $LP_{i,a}$. Let $|PP_a|$ be the number of LPs mapped on PP_a , i.e., the number of LPs in a scheduling queue of PP_a . For the inter-processor LPs, the null message cost is the point-to-point message transmission time as usually known as remote null message cost, however, intra-processor LPs pay for the (local) null message cost which is much less than the remote null message cost. For the intra-processor null message cost, we assume the cost is zero since it is no other than a local memory access.

For an $LP_{i,a}$ and its critical parent $LP_{critical,b}$, the physical time of EIT update is: $T(eit_i) = T(eot_{critical}) + T_m = T(eit_{critical}) + T_m$ if $a \neq b$.

There should be another consideration of processing time of other local LPs since we assume several LPs are mapped on one CPU. Assume that the critical LP and its child are on the same processor. If the critical parent of $LP_{i,a}$, $LP_{critical,a}$ is the first to be scheduled on PP_a after the processor being idle, the EIT update time is $T(eit_i) = T(eot_{critical}) = T(eit_{critical})$ since we only need local null message cost. Otherwise, $T(eit_i) = T(eit_{critical}) + \sum_{l \neq i, \forall LP_{l,a} \text{ on } PP_a} \overline{T}_l \times n_l^{safe}$ if $LP_{i,a}$ is the last LP scheduled among the LPs on PP_a . Note that n_l^{safe} stands for the number of safe message on $LP_{l,a}$ at that time when LP is being scheduled, and \overline{T}_l is the average time for an LP_l to process an event. The detailed discussion on the number of safe messages can be found in [11]. By scheduling LPs with CPPS, we can force a probable critical LP to be scheduled first among the LPs mapped on one CPU. The processing

of LPs on PP_a prior to the critical $LP_{i,a}$ is the key factor of scheduling analysis. Regardless of whether the critical parent and its child are on the same processor or not, the CPPS mechanism can guarantee that a critical LP always precedes the other LPs on the same processor to process its events.

The benefit of CPPS over URS follows. For the URS, scheduler randomly chooses next LP to be scheduled without any preference of LPs therefore the average number of LPs scheduled in front of the critical LP is $0.5 \times |PP_a|$. Let

$$p(LP_{i,a}, LP_{child,b}) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \text{ or } LP_{child,b} \notin DS_i \end{cases}$$

be a boolean function which indicates if $LP_{i,a}$ is on the same processor to one of its successor, $LP_{child,b}$.

A critical LP must wait the other LPs on the same processor before it is scheduled by URS. For an arbitrary $LP_{i,b}$ with the critical parent $LP_{critical,a}$, the physical time of an EIT update is:

$$\begin{aligned} T(eit_i) &= T(eit_{critical}) + T_m \times p(LP_{critical,a}, LP_{i,b}) \\ &\quad + 0.5 \times \sum_{l / \nexists \forall LP_{l,a} \text{ on } PP_a} \overline{T}_l \times n_l^{safe} \end{aligned} \quad (1)$$

However, the CPPS guarantees:

$$T(eit_i) = T(eit_{critical}) + T_m \times p(LP_{critical,a}, LP_{i,b}) \quad (2)$$

The effect of critical LP scheduling can be chained from the source LP to a specific LP over the precedence critical path. Considering the chains of critical LPs on a PCP of LP_i , (1) and (2) can be expanded as:

$$\begin{aligned} (a) \quad T(eit_i) &= T(eot_{source}) + \sum_{LP_j \in PCP_i} \{p(LP_{j,b}, LP_{child,c}) \times T_m \\ &\quad + 0.5 \times \sum_{k / \nexists \forall LP_{k,b} \text{ on } PP_b} \overline{T}_k \times n_k^{safe}\} \quad \text{for the URS} \\ (b) \quad T(eit_i) &= T(eot_{source}) + \sum_{LP_j \in PCP_i} \{p(LP_{j,b}, LP_{child,c}) \times T_m\} \\ &\quad \text{for the CPPS} \end{aligned} \quad (3)$$

The rate of an acceleration of an EIT update (as we termed benefit) of the CPPS over the URS can be drawn from (3)-a and (3)-b.

$$\begin{aligned} Benefit &= \\ &= \frac{T(eot_{source}) + \sum_{LP_j \in PCP_i} p(LP_{j,b}, LP_{child,c}) \times T_m}{T(eot_{source}) + \sum_{LP_j \in PCP_i} \{p(LP_{j,b}, LP_{child,c}) \times T_m + 0.5 \times \sum_{k \neq i, \forall LP_{k,b} \text{ on } PP_b} \overline{T}_k \times n_k^{safe}\}} \end{aligned} \quad (4)$$

3.2 Non-critical Null Message Reduction

Equation (2) implies that a critical parent can also send its EOT to successors without waiting other LPs to process their events. Thus it will repeatedly accelerate the child LP's EIT update time regardless whether the child LP is critical

or not. In addition, null messages from the critical parent can be always critical, i.e., it can always update the child's EIT. It is clear that we can execute the same conservative parallel simulation with smaller number of null messages with the CPPS since we can minimize the non-critical null messages. The reduced number of non-critical null messages with CPPS over those with URS is another factor of scheduling benefit. The minimum number of null messages for a conservative simulation can be found in [8] which is also the minimum bound for CPPS. For the URS, non-critical LPs scheduled prior to critical parent could send non-critical null message despite a non-critical null message for an LP might be a pseudo critical one for the other LPs. However, CPPS guarantees that there can be no null message sent before the critical null message.

3.3 Total Scheduling Benefit

Based on the above two subsections, the benefit of scheduling comes from two factors. The first factor is the accelerated EIT update. The second factor is reduction of trivial null messages by sending critical null message first. Incorporating the reduction of non-critical null messages to (4), the total scheduling benefit is:

$$\begin{aligned} Benefit = & [T(eot_{source}) + \sum_{\exists PP_b, PP_c, LP_j \in PCP_i} p(LP_{j,b}, LP_{child(j),c}) \times T_m] / \\ & [T(eot_{source}) + \sum_{\exists PP_b, PP_c, LP_j \in PCP_i} \{p(LP_{j,b}, LP_{child(j),c}) \times T_m \\ & + 0.5 \times \sum_{k \neq i, \forall LP_{k,b} \text{ on } PP_b} (\bar{T}_k \times n_k^{safe} + T_m)\}] \end{aligned} \quad (5)$$

where $LP_{child(j)}$ stands for a successor LP of LP_j on the critical path. We assumed a source LP sends EOT at $T(eot_{source})$ for a specific EIT update. With the usual fact that CPU is far faster than interconnection network, we approximate that $\bar{T}_k \times n_k^{safe} \ll T_m$ holds for any LP_k to remove the factor of number of safe messages. The final result of scheduling benefit is:

$$\begin{aligned} Benefit = & [T(eot_{source}) + \sum_{\exists PP_b, PP_c, LP_j \in PCP_i} p(LP_{j,b}, LP_{child(j),c}) \times T_m] / \\ & [T(eot_{source}) + \sum_{\exists PP_b, PP_c, LP_j \in PCP_i} \{p(LP_{j,b}, LP_{child(j),c}) \times T_m \\ & + 0.5 \times \sum_{k \neq i, \forall LP_{k,b} \text{ on } PP_b} T_m\}] \end{aligned} \quad (6)$$

4 Experimental Validation

4.1 Design of Experiments

We introduce several experiments to validate the effectiveness of the mathematical analysis. We use two synthetic simulation models, CQNF-GRID and CQNF-TANDEM as described in [5], for the validation under the similar environment. Simulation run-time system for the URS basically uses an unsorted queue for schedulers (The same copy of scheduler resides on each processors.) in its conservative algorithms, while event scheduling queue for every LP is a splay tree. The CPPS version of simulation run-time system has the LP scheduling as a splay

tree. Event messages and null messages are transmitted over MPI library on distributed systems or pthread on shared memory multiprocessors. In our runtime, we implemented non-preemptive scheduling since each LP must yield CPU voluntarily once it is blocked. Experiments were performed on a Sun SPARC 1000 server with eight processors and 512MB main memory. We set two groups of experiments for validation. The first is a basic experiment of CQNF-TANDEM to see the effect of load balancing with CPPS. Since time-variant critical parent can be partitioned on several processors, the balanced load is important because it determines how many LPs the critical parent will wait. To see the effect of load balancing, varying number of processors with the constant number of LPs are configured and then fixed number of processors (four) and increasing number of LPs are configured. The second is an experiment of CQNF-GRID with varying connectivity. With denser connectivity, an LP must send more null messages so that we expect more speed-up with CPPS.

4.2 Effect of Load Balancing

Figure 1-a shows the change in the speed-up of the CQNF-Tandem benchmark and analytical result from (6). The experiment used 192 LPs in total. Note that by fixing the total number of LPs in the model, the number of LPs per processor decreases as the number of processors increases.

The analysis shows a positive effect even with one processor, but the experiment shows a negative effect. This phenomenon is due to the scheduling queue overhead because of the addition of splay tree to run-time system with $O(\log n)$ insertion overhead. It also explains the gap between analytical and experimental results. With two or more processors, the scheduling benefit is distinct in both curves. In three-, five- and six-processor cases, each two-processor has one extra tandem queue. In seven-processor case, each four-processor has one extra tandem queue. With these load imbalances, experimental result shows larger LP scheduling queue overhead while analytical results attenuate the impact of load imbalance. Figure 1-b shows the results of an experiment where the number of processors is held at four, but the number of LPs per processor was varied. LPs

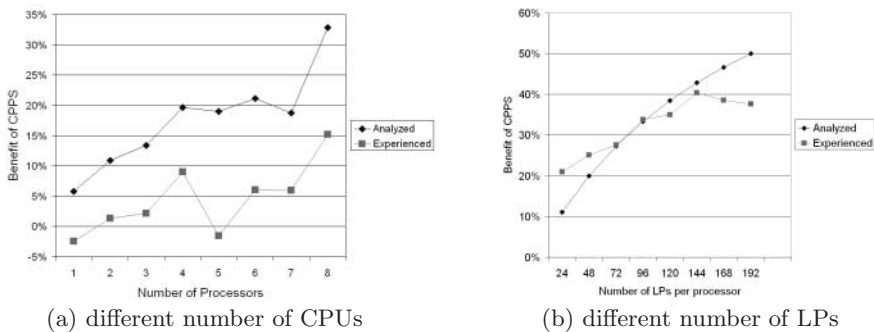


Fig. 1. Effect of load balancing

can be added by adding new tandem queues (six LPs at each queue). All of the routers are completely connected to the other tandem queues, so the communication density also increases. The analysis predicts the speed-up increases as the number of LPs uniformly increases. With smaller number of LPs the speed-up is higher in the experiment than in the analysis, since saturation of splay tree management overhead degrades the effect of scheduling with huge number of LPs on a CPU.

4.3 Effect of Connectivity

Increasing the communication definitely brings higher overhead in the null message protocol. With two-dimensional grid topology, CQNF-GRID, an LP has more complicated PCP with increasing connectivity. From the view of (6), complicated PCP increases the scheduling benefit of CPPS.

Figure 2 shows the benefit of CPPS over URS according to the number of neighbors. There are fixed number of LPs (382 LPs) on four processors in each experiment to harmonize the effect of scheduling queue overhead and to balance the load. With smaller connectivity the effect of CPPS remains almost same, however higher connectivity shows the abrupt increase of CPPS benefit both in analyzed and experienced results.

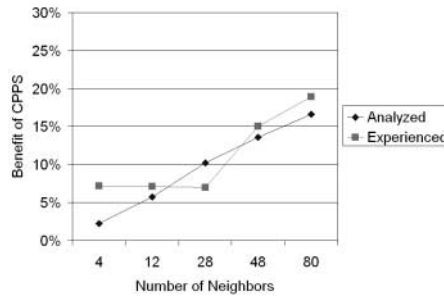


Fig. 2. Effect of CPPS with variable connectivity

5 Conclusion

The optimal scheduling strategy for a parallel discrete event simulation is to schedule events on the critical path first (assigning independent overlapping events to other processors). The LP with the minimum EOT might not have the critical event, but it is also critical in a sense that it is limiting the progress of all its descendents, and should therefore be preferentially scheduled. With this key idea, CPPS was introduced and the mathematical analysis of CPPS benefit has been demonstrated in this research.

The experimental results validate our asymptotic analysis of CPPS is useful. Even though the effect of CPPS is clear in (6), the results in section 4 imply that there is a room to improve the conservative scheduling on the implementation level. We underestimated the scheduling overhead of splay queue management. One of the experimental results shows abnormality because of scheduling overhead. (In figure 1-b, analyzed benefit is larger than experienced ones sometimes.) We assumed the interconnection network used in experiments are ideal to send timely null messages, while experimental result implies the interconnection network is sometimes saturated with huge number of LPs.

Further study is required to improve the analytical model of scheduling performance. The processing of safe messages requires a queueing model to be introduced and there needs a more sophisticated model of interconnection network. Combining this study with good delay model of LP scheduling queue will bring the best result in scheduling analysis. The lookahead is very well known as a huge factor of performance in conservative simulation. Clearly, CPPS has benefit even with poor lookaheaded simulation models. The analytical model of scheduling benefit incurred with lookahead is considerable.

References

- [1] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, 1981. 191
- [2] D. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, July 1985. 191
- [3] S. Srinivasan and Jr. P. F. Reynolds. Super-criticality revisited. In *11th Workshop on Parallel and Distributed Simulation*, Lake Placid, NY, USA, 1995. 191
- [4] E. Naroska and U. Schwiegelshohn. A new scheduling method for parallel discrete-event simulation. In *Second Int'l Euro-Par Conference*, Lyon, France, 1996. 192
- [5] H. Y. Song, R. A. Meyer, and R. L. Bagrodia. An empirical study of conservative scheduling. In *PADS2000*, Bologna, Italy, 2000. 192, 195
- [6] et al Xiao, Z. Scheduling critical channels in conservative parallel discrete event simulation. In *PADS99*, Atlanta, Georgia, USA, 1999. 192
- [7] J. Liu, D. M. Nicol, and K. Tan. Lock-free scheduling of logical processes in parallel simulation. In *PADS01*, Lake Arrowhead, CA, USA, 2001. 192
- [8] H. Y. Song, S. H. Cho, and S. Y. Han. A null message count for conservative parallel simulation. In *VecPar 2002*, Porto, Portugal, 2002. 192, 195
- [9] R. Roenngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM TOMACS*, 7(2):157–209, 1997. 193
- [10] R. Roenngren and R. Ayani. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–86, 1985. 193
- [11] H. Y. Song. A probabilistic performance model for conservative simulation protocol. In *PADS01*, Lake Arrowhead, CA, USA, 2001. 193

Parallel Computation of Pseudospectra Using Transfer Functions on a MATLAB-MPI Cluster Platform

Constantine Bekas¹, Efrosini Kokiopoulou¹,
Efstratios Gallopoulos¹, and Valeria Simoncini²

¹ Computer Engineering & Informatics Department, University of Patras, Greece
`{knb,exk,statis}@hpcclab.ceid.upatras.gr`

² Mathematics Department, University of Bologna, Italy
`val@dragon.ian.pv.cnr.it`

Abstract. One of the most computationally expensive problems in numerical linear algebra is the computation of the ϵ -pseudospectrum of matrices, that is, the locus of eigenvalues of all matrices of the form $A + E$, where $\|E\| \leq \epsilon$. Several research efforts have been attempting to make the problem tractable by means of better algorithms and utilization of all possible computational resources. One common goal is to bring to users the power to extract pseudospectrum information from their applications, on the computational environments they generally use, at a cost that is sufficiently low to render these computations routine. To this end, we investigate a scheme based on *i*) iterative methods for computing pseudospectra via approximations of the resolvent norm, with *ii*) a computational platform based on a cluster of PCs and *iii*) a programming environment based on MATLAB enhanced with MPI functionality and show that it can achieve high performance for problems of significant size.

1 Introduction and Motivation

Let $A \in \mathbb{C}^{n \times n}$ have singular value decomposition (SVD) $A = U\Sigma V^*$, and let $\Lambda(A)$ be the set of eigenvalues of A . The ϵ -pseudospectrum $\Lambda_\epsilon(A)$ (pseudospectrum for short) of a matrix is the locus of eigenvalues of $\Lambda(A + E)$, for all possible E such that $\|E\| \leq \epsilon$ for given ϵ and norm. When A is normal (i.e. $AA^* = A^*A$), the pseudospectral regions are readily computed as the union of the disks of radius ϵ surrounding each eigenvalue of A . When A is nonnormal, the pseudospectrum is no longer readily available and we need to approximate it. As has been discussed elsewhere, the pseudospectrum frequently provides important model information that is not available from the matrix spectrum, e.g. for the behavior of iterative methods on large, sparse nonnormal matrices; see the bibliography at web.comlab.ox.ac.uk/projects/pseudospectra, [1, 2, 3] as well as [4] for a related PVM package. An important barrier in providing pseudospectral information routinely is the expense involved in their calculation. In the sequel we use the 2-norm for matrices and vectors, $R(z) = (A - zI)^{-1}$ is the

resolvent and e_k the k^{th} standard unit vector. Also $s(x, y) := \sigma_{\min}(xI + iyI - A)$, herein written compactly as $s(z)$ for $z = x + iy$, is the minimum singular value of matrix $zI - A$. Two equivalent definitions of the ϵ -pseudospectrum are

$$\Lambda_\epsilon(A) = \{z \in \mathbb{C} : \sigma_{\min}(A - zI) \leq \epsilon\} = \{z \in \mathbb{C} : \|R(z)\| \geq \frac{1}{\epsilon}\}. \quad (1)$$

Relations (1) immediately suggest an algorithm, referred to as **GRID**, for the computation of $\Lambda_\epsilon(A)$ that consists of the following steps: *i*) Define grid Ω_h on a region of the complex plane that contains $\Lambda_\epsilon(A)$. *ii*) Compute $s(z_h) := \sigma_{\min}(z_h I - A)$ or $R(z_h)$, $\forall z_h \in \Omega_h$. *iii*) Plot ϵ -contours of $s(z)$ or $R(z)$. **GRID**'s cost is modeled as

$$C_{\text{GRID}} = |\Omega_h| \times C_{\sigma_{\min}}, \quad (2)$$

where $|\Omega_h|$ is the number of nodes of the mesh and $C_{\sigma_{\min}}$ is the average cost of computing σ_{\min} . **GRID** is simple, robust and embarassingly parallel, since the computations of $s(z_h)$ are completely decoupled between gridpoints. As cost formula (2) reveals, its cost increases rapidly with the size of the matrix and the number of gridpoints in Ω_h . For example, using **MATLAB** version 6.1, the **LAPACK** 3.0 based **svd** function takes 19 sec to compute the SVD of a 1000×1000 matrix on a Pentium-III at 866 MHz. Computing the pseudospectrum on a 50×50 mesh would take almost 13 hours or about 7, if A is real and we exploit the resulting symmetry of the pseudospectrum with respect to the real axis. Exploiting the embarassingly parallel nature of the algorithm, it would take over 150 such processors to bring the time down to 5 min (or 2.5, in case of real A). As an indication of the state of affairs when iterative methods are used, **MATLAB**'s **svds** (based on **ARPACK**) on the same machine required almost 68 sec to compute $s(z)$ for Harwell-Boeing matrix **e30r0100** (order $n = 9661$, **nnz** = 306002 non-zero elements) and random z ; furthermore, iterative methods over domains entail interesting load balancing issues; cf. [5, 6]. The above underline the fact that computing pseudospectra presents a formidable computational challenge since typical matrix sizes in applications can reach the order of hundreds of thousands. Cost formula (2) suggests that we can reduce the complexity of the calculation by *domain-based* methods, that seek to reduce the number of gridpoints where $s(z)$ must be computed, and *matrix-based* methods, that target at reducing the cost of evaluating $s(z)$.

Given the complexity of the problem, it also becomes necessary that we use any advances in high performance computing at our disposal. All of the above approaches are the subject of current research.¹ Therefore, if the pseudospectrum is to become a practical tool, we must bring it to the users on the computational environments they generally use, at an overall cost that is sufficiently low to render these computations routine. Here we present our efforts towards this target and contribute a strategy that combines: Parallel computing on clusters of PCs for high performance at low cost, communication using MPI-like calls, and programming using **MATLAB**, for rapid prototyping over high quality libraries with

¹ See also the site URL <http://web.comlab.ox.ac.uk/projects/pseudospectra>.

a very popular system. Central to our strategy is the Cornell Multitasking Toolbox for MATLAB (CMTM) ([7]) that enhances MATLAB with MPI functionality [8]. In the context of our efforts so far, we follow here the “transfer function framework” described in [9]. Some first results, relating the performance of the transfer function framework on an 8 processor Origin-2000 ccNUMA system were presented in [10]. In [5], we described the first use of the CMTM system in the context of domain-based algorithms developed by our group as well as some load balancing issues that arise in the context of domain and matrix methods. In this paper, we extend our efforts related to the iterative computation of pseudospectra using the *transfer function framework*.

1.1 Computational Environment

As a representative of the kind of computational cluster that many users will not have much difficulty in utilizing, in this paper we employ up to 8 uniprocessor Pentium-III @ 933MHz PCs, running Windows 2000, with 256KB cache and 256MB memory, connected using a fast Ethernet switch. We use MATLAB (v. 6.1) due to the high quality of its numerics and interface characteristics as well as its widespread use in the scientific community. The question arises, then, how to use MATLAB to solve problems such as the pseudospectrum computation in parallel? We can, for example translate MATLAB programs into source code for high performance compilers e.g. [11], or use MATLAB as interface for calls to distributed numerical libraries e.g. [12]. Another approach is to use multiple, independent MATLAB sessions with software to coordinate data exchange. This latter approach is followed in the “Cornell Multitasking Toolbox for MATLAB” [7], a descendant of the MultiMATLAB package [13]. CMTM enables multiple copies of MATLAB to run simultaneously and to exchange data arrays via MPI-like calls. CMTM provides easy-to-use MPI-like calls of the form `A = MMPI_Bcast(A);` Each MMPI function is a MATLAB mex file, linked as a DLL with a commercial version of MPI, MPI/Pro v. 1.5, see [14]. Like MATLAB, CMTM functions take as arguments arrays. As the above broadcast command shows, CMTM calls are simpler than their MPI counterparts. The particular command is executed by all nodes; the system automatically distinguishes whether a node is a sender or a receiver and performs the communication. Similar are the calls to other communication procedures such as blocking send-receive and reduction clauses. For example, in order to compute a dot product of two vectors x and y that are distributed among the processors, all nodes execute the call `MMPI_Reduce(x'*y, MPI_ADD);` Each node performs the local dot product and then the system accumulates all local sub-products to a root node. Table 1 lists the commands available in the current version of CMTM. Notice the absence of the MPI ALL type commands (e.g. `Allreduce`, `Allgather`). Whenever these commands are needed, we simulate them, e.g. using reduce or gather followed by broadcast.

Baseline measurements In order to better appreciate the performance of our cluster and the results for our main algorithm, we provide baseline measurements for selected collective communication operations. All experiments in this work

Table 1. Available CMTM commands (version 0.82) (actual commands prefixed with MMPI).

| |
|---|
| Abort, Barrier, Bcast, Comm_dup, Comm_free, Comm_rank |
| Comm_size, Comm_split, Gather, Iprobe, Irecv, Recv |
| Reduce, Scatter, Scatterv, Send, Testany, Wtime |

Table 2. Timings (in sec) for broadcasting and reducing summation of vectors of size 40000:20000:100000 using CMTM on $p = 2, 4$ and 8 processors

| | MMPI_Bcast | | | | MMPI_Reduce | | | |
|-----|------------|-------|-------|--------|-------------|-------|-------|--------|
| p | 40000 | 60000 | 80000 | 100000 | 40000 | 60000 | 80000 | 100000 |
| 2 | 0.03 | 0.05 | 0.06 | 0.08 | 0.05 | 0.05 | 0.06 | 0.081 |
| 4 | 0.07 | 0.09 | 0.12 | 0.15 | 0.09 | 0.13 | 0.13 | 0.16 |
| 8 | 0.09 | 0.13 | 0.17 | 0.22 | 0.13 | 0.15 | 0.19 | 0.24 |

were conducted in single user mode. Table 2 contains the measured timings with varying number of processors and problem sizes when using CMTM to broadcast vectors and also to reduce by summation. As expected, `MMPI_Reduce` is always more time consuming than `MMPI_Bcast`, while both take longer as the cluster size increases.

2 The Transfer Function Framework

An early, interesting idea for the economical approximation of the pseudospectrum, presented in [15], was to use Krylov methods and obtain approximations to $\sigma_{\min}(zI - A)$, from $\sigma_{\min}(zI - H)$, where H was either the square or the rectangular upper Hessenberg matrix that is obtained during the Arnoldi iteration and I is the identity matrix or a section thereof to conform in size with H . A practical advantage of this approach is that, whatever the number of gridpoints, it requires only one (expensive) transformation and compression (when $m < n$) of A to upper Hessenberg form. It still requires, however, the (independent) computations of $\sigma_{\min}(z_h I - H)$, for every gridpoint z_h , since, unlike eigenvalues singular values do not respect the shift property, i.e. typically $\sigma(zI - A) \neq -\sigma(A)$. The above approach was further refined in [16]. Let now D^*, E be full rank - typically long and thin - matrices, of row dimension n . Matrix $G_z(A, E, D) := DR(z)E$ is called in the literature *transfer function*. It was shown in [9] that the transfer function provides a useful framework for describing existing and defining new methods for computing pseudospectra. The central method in [9] was based on the selection $D = W_m^*$ and $E = W_{m+1}$, where $W_m = [w_1, \dots, w_m]$ denotes the orthonormal basis for the Krylov subspace $\mathcal{K}_m(A, w_1)$ constructed by the

Arnoldi iteration, so that

$$A W_m = W_m H_{m,m} + h_{m+1,m} w_{m+1} e_m^\top, \quad (3)$$

where $H_{m,m}$ is the square upper Hessenberg matrix consisting of the first m rows of $H_{m+1,m}$. We note that $W_{m+1} = [W_m, w_{m+1}]$ and $H_{m+1,m} = [H_{m,m}; h_{m+1,m} e_m^\top]$. We will be referring to m as the “transfer function dimension” and will be writing $G_{z,m}(A)$ for $G_z(A, W_{m+1}, W_m^*)$. With the aforementioned selection for D and E , for large enough m we have $\|R(z)\| \approx \|G_z(A, W_{m+1}, W_m^*)\|$, which provides an approximation to $\|R(z)\|$ that improves monotonically with m and would be used in the sequel to construct the pseudospectrum based on relation (1); cf. [9]. Even though computation of $G_z(A, W_{m+1}, W_m^*) = W_m^* (A - zI)^{-1} W_{m+1}$ appears to require solving $m+1$ linear systems of size n , a trick allows us to reduce the number of (large) linear systems to one because

$$G_{z,m}(A) = [(\tilde{I} - h_{m+1,m} \phi_z e_m^\top)(H_{m,m} - zI)^{-1}, \phi_z], \quad (4)$$

where $\phi_z = W_m^* (A - zI)^{-1} w_{m+1}$; cf. [9]. Therefore, in order to compute $\|G_{z,m}(A)\|$ we have to solve a single (large) linear system of size n for each shift z ; furthermore, the right hand side, w_{m+1} , remains the same for each shift, something that will be exploited by the iterative solver (cf. the next section). We also need to solve m Hessenberg systems of size m , and to compute the norm of $G_{z,m}(A)$. We discuss the actual parallel implementation of this approach for our cluster environment in Section 2.1.

The Arnoldi iteration As with most modern iterative schemes, the effective implementation of the transfer function methodology uses the Arnoldi iteration, via an implementation of relation (3), as a computational kernel, to create an orthogonal basis for the Krylov subspace [17]. We organize the iteration around a row-wise partition of the data: Each processor is assigned a number of rows of A and V . At step j , we store the entire $V(:, j)$ that must be orthogonalized into each processor. This requires a broadcast, at some point during the iteration, but allows thereafter the matrix-vector multiplication to take place locally on each processor. Due to lack of space we do not go into further details but tabulate, in Table 3, results from experiments on our cluster and programming environment with three different types of Arnoldi iteration: CGS-Arnoldi, MGS-Arnoldi and CGSo-Arnoldi, where the first is based on classical Gram-Schmidt (GS) orthogonalization, the second on modified GS and the third on classical GS with selective reorthogonalization. Notice the superlinear speedup for the larger matrices due to the distribution of the large Krylov bases. The random sparse matrices were created using the MATLAB’s built-in function `sprand`(n, n, ρ) with ρ equal to 3×10^{-4} , that is $n \times n$ matrices with approximately $\rho * n^2$ uniformly distributed nonzero entries.

2.1 TRGRID: GRID with Transfer Functions

The property that renders Krylov type linear solvers particularly suitable in the context of the transfer function framework for pseudospectra, in particu-

Table 3. Runtimes (in sec) for three versions of Arnoldi orthogonalization on $p = 1$ to 8 processors on random sparse matrices of size n

| Arnoldi | $p \setminus n$ | 40000 | 60000 | 80000 | 100000 | $p \setminus n$ | 40000 | 60000 | 80000 | 100000 |
|---------|-----------------|-------|-------|-------|--------|-----------------|-------|-------|-------|--------|
| CGS | 1 | 328 | 512 | 1716 | 5885 | 2 | 150 | 241 | 340 | 440 |
| MGS | | 362 | 560 | 1453 | 3126 | | 172 | 265 | 365 | 476 |
| CGSo | | 389 | 606 | 1780 | 6962 | | 181 | 289 | 405 | 522 |
| CGS | 4 | 80 | 126 | 179 | 235 | 8 | 53 | 80 | 113 | 143 |
| MGS | | 129 | 158 | 209 | 268 | | 92 | 125 | 188 | 201 |
| CGSo | | 101 | 150 | 211 | 275 | | 64 | 97 | 135 | 164 |

lar formula (4), is the shift invariance of Krylov subspaces, i.e. the fact that $\mathcal{K}_d(A, w_1) = \mathcal{K}_d(A - zI, w_1)$ for any z . This means that we can reuse the same basis that we build for $\mathcal{K}_d(A, w_1)$, say, to solve linear systems with “shifted” matrices, e.g. $A - zI$; see [18]. Since our interest is in nonnormal matrices, we used **GMRES** as the linear solver. The pseudocode for our parallel algorithm is listed in Table 4.

Remember that after completing the (parallel) Arnoldi iterations (Table 4, line 1), with the row distribution described earlier, the rows of the bases W and \hat{W} are distributed among the cluster nodes. For the sake of economy in notation we use the same symbolism when we deal with local computations too, while in reality we are referring only to a subset of the rows of W and \hat{W} . Each processor (see lines 2-4, 9-12) works on M/p gridpoints (we have assumed that p exactly divides M). In line 3, each node computes its share of the columns of matrix Y . For the next steps, in order for the local ϕ_z to be computed, all processors must access the whole Y , hence all-to-all type communication is needed. In lines 9-12 each processor works on its share of gridpoints and columns of matrix Φ_z . Finally, processor zero gathers the approximations of $\{1/s(z_i)\}_{i=1}^M$. What remains to clarify is the computation of $\phi_{z,i}$ and Φ_z (lines 6-7). We have $\phi_{z,i} = W_i^* \hat{W}_i Y$, where W_i and $\hat{W}_i, i = 0 \dots, P - 1$ are the subsets of contiguous rows of the basis matrices W and \hat{W} that each node privately owns. Then, $\Phi_z = \sum_{i=0}^{P-1} \phi_{z,i}$ and we use reducing summation to compute Φ_z in parallel, since **CMTM** allows reduction processes to work on matrices as well as scalars. It is worth noting that the preferred order used when applying to implement the two matrix-matrix multiplications of line 6 depends on the dimensions of the local matrices W_i, \hat{W}_i and on the column dimension of Y and could be decided at run time. In our experiments, we assumed that the Krylov dimensions m, d were much smaller than the size of the matrix, n , and the grid, M . **TRGRID** consists of two parts. The first part (line 1) consists of the Arnoldi iterations used to obtain orthogonal bases for the Krylov subspaces used for the transfer function and for the solution of the size n system. In forthcoming work we detail techniques that exploit the common information present in the two bases ([10]). An important advantage of the method is that after the first part is completed and basis matrices W and \hat{W} become available, we only need to repeat the

Table 4. Pseudocode for the parallel TRGRID algorithm for the CMTM environment

```

Parallel TRGRID
(* Input *) Points  $z_i, i = 1, \dots, M$ , vector  $w_1$ 
           with  $\|w_1\| = 1$ , scalars  $m, d$ .
1. All nodes work on two parallel Arnoldi iterations
    $[W_{m+1}, H_{m+1,m}] \leftarrow \text{arnoldi}(A, w_1, m)$ 
    $[\hat{W}_{d+1}, F_{d+1,d}] \leftarrow \text{arnoldi}(A, w_{m+1}, d)$ 
   Each node has some rows of  $W_{m+1}, \hat{W}_{d+1}$ 
   Node zero distributes the  $M$  gridpoints
   so that processor  $pr\_id$  holds points in the  $M/p$ -sized set  $I(pr\_id)$ 
2. for  $i \in I(pr\_id)$ 
3.    $Y(:, i) = \text{argmin}_y \{ \|(F_{d+1,d} - z_i \tilde{I}_d)y - e_1\| \}$ 
4. end
5. Node zero gathers matrix  $Y$  and broadcasts it back
6. Each node performs local multiplication  $\phi_{z,i} = W_m^* \hat{W}_d Y$ 
7. Node zero collects  $\Phi_z = \text{Reduce}(\phi_z, \text{SUM})$ 
8. Node zero distributes back the columns of  $\Phi_z$ 
9. for  $i \in I(pr\_id)$ 
10.   $D_i = (\tilde{I} - h_{m+1,m} \Phi_z(:, i) e_m^\top)(H_{m,m} - z_i I)^{-1}$ 
11.   $\|G_{z_i}(A)\| = \|[D_i, \phi_{z_i}]\|$ 
12. end
13. Node zero gathers approximations of  $1/s(z_i)$ 

```

second part, from line 2 onwards to compute the pseudospectrum. Based on this property and given the previous results for Arnoldi we measure the performance of only the second part of TRGRID. A second observation is that the second TRGRID involves only dense linear algebra. Remember also that at the end of the first part, each processor has a copy of the upper Hessenberg matrices. Hence, we can use BLAS-3 to exploit the memory hierarchy of each processor. Furthermore, assuming that $M \geq p$, the workload would be evenly divided amongst processors. Of course, the method is always subject to load imbalances due to systemic effects. However, since the second part of parallel TRGRID requires only limited communication, increased network traffic is not expected to have a significant effect on performance. Let us now investigate the performance of the main body of parallel TRGRID on our cluster's nodes. We experimented with the HB-matrices **gre_1107** ($n = 1107, \text{nnz} = 5664$) and **pores_2** ($n = 1124, \text{nnz} = 9613$) scaled by $1e-7$, as well as with two random sparse matrices of size $n = 2000$ and 4000 with density $\rho = 0.01$. On the domain $[-1.8, 1.8] \times [-1.8, 1.8]$, we employed a 50×50 grid for the HB matrices and a 30×40 grid for the random matrices. Figure 1 illustrates the $\epsilon = 0.1$ contours for the two HB matrices. The Krylov dimensions were $m = d = 75, 100, 120$ and 150 respectively. Table 5 illustrates the performance results. In all cases, except for the random matrices when $p = 8$

Table 5. Performance of parallel TRGRID

| | gre_1107 | | | | pores_2 | | | | sprand(2000) | | | | sprand(4000) | | | |
|------------|----------|------|------|------|---------|-----|------|------|--------------|-------|------|------|--------------|------|-----|------|
| p | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 |
| Time (sec) | 174 | 76.2 | 36.2 | 18.7 | 340 | 155 | 76.3 | 38.1 | 244 | 118.8 | 59.3 | 31 | 501 | 245 | 123 | 63.9 |
| Speedup | - | 2.3 | 4.8 | 9.3 | - | 2.2 | 4.5 | 8.9 | - | 2.1 | 4.1 | 7.87 | - | 2.04 | 4.1 | 7.84 |

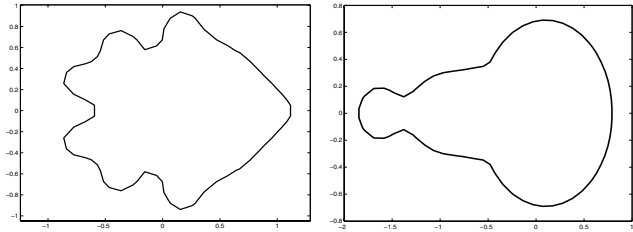


Fig. 1. Pseudospectrum contour $\partial\Lambda_\epsilon(A)$, $\epsilon = 1e - 1$ for `gre_1107` (left) and `pores_2` (right)

are used, we observe typically superlinear absolute speedups due to the fact that the Krylov bases are distributed.

3 Conclusions

The widespread availability of the high quality numerics and interface of `MATLAB` together with an MPI-based parallel processing toolbox and the results shown in this paper for matrix-based methods and in [5] for domain based methods, indicate that the computation of pseudospectra of large matrices is rapidly becoming accessible to everyday users who prefer to program in `MATLAB` and whose only hardware infrastructure is a PC cluster. Further work in progress includes the incorporation of these methods into a toolbox, the exploitation of multi-threading, enhancement of the numerical infrastructure with lower level parallel computational kernels, e.g. `SCALAPACK`-like and, finally, resolving the numerical issues that arise when combining the transfer function approach with domain based methods (in line with [10]).

Acknowledgments

We thank the referees and Dr. Mark Embree for their comments and help. The first author wishes to thank the Bodossaki Foundation for its support.

References

- [1] Trefethen, L., Trefethen, A., Reddy, S., Driscoll, T.: Hydrodynamic stability without eigenvalues. *Science* **261** (July 1993) 578–584 [199](#)
- [2] Trefethen, L.: Pseudospectra of linear operators. *SIAM Rev.* **39** (1997) 383–406 [199](#)
- [3] Trefethen, L.: Computation of pseudospectra. In: *Acta Numerica 1999*. Volume 8. Cambridge University Press (1999) 247–295 [199](#)
- [4] Braconnier, T.: Fvpspack: A Fortran and PVM package to compute the field of values and pseudospectra of large matrices. Numerical Analysis Report No. 293, Manchester Centre for Computational Mathematics, Manchester, England (1996) [199](#)
- [5] Bekas, C., Kokiopoulou, E., Koutis, I., Gallopoulos, E.: Towards the effective parallel computation of matrix pseudospectra. In: *Proc. 15th ACM Int'l. Conf. Supercomputing (ICS'01)*, Sorrento, Italy. (2001) 260–269 [200](#), [201](#), [206](#)
- [6] Mezher, D., Philippe, B.: Parallel computation of the pseudospectrum of large matrices. *Parallel Computing* **28** (2002) 199–221 [200](#)
- [7] Zollweg, J., Verma, A.: The Cornell Multitask Toolbox. Directory Services/Software/CMTM at URL <http://www.tc.cornell.edu> (2002) [201](#)
- [8] Gropp, W., Lusk, E., Skjellum, A.: *Using MPI: Portable Parallel Programming with the Message Programming Interface*. second edn. MIT Press, Cambridge, MA (1999) [201](#)
- [9] Simoncini, V., Gallopoulos, E.: Transfer functions and resolvent norm approximation of large matrices. *Electronic Transactions on Numerical Analysis (ETNA)* **7** (1998) 190–201 [201](#), [202](#), [203](#)
- [10] Bekas, C., Gallopoulos, E., Simoncini, V.: On the computational effectiveness of transfer function approximations to the matrix pseudospectrum. In: *Proc. Copper Mountain Conf. Iterative Methods*. (Apr. 2000) [201](#), [204](#), [206](#)
- [11] DeRose, L., et al.: FALCON: A MATLAB Interactive Restructuring Compiler. In C.-H. Huang, et al., ed.: *LNCS: Languages and Compilers for Parallel Computing*. Springer-Verlag, New York (1995) 269–288 [201](#)
- [12] Casanova, H., Dongarra, J.: NetSolve: A network server for solving computational science problems. *Int'l. J. Supercomput. Appl. and High Perf. Comput.* **11** (1997) 212–223 [201](#)
- [13] Menon, V., Trefethen, A.: MultiMATLAB: Integrating MATLAB with high-performance parallel computing. In: *Proc. Supercomputing'97*. (1997) [201](#)
- [14] MPI Software Technology: (MPI/Pro) <http://www.mpi-softtech.com>. [201](#)
- [15] Toh, K. C., Trefethen, L.: Calculation of pseudospectra by the Arnoldi iteration. *SIAM J. Sci. Comput.* **17** (1996) 1–15 [202](#)
- [16] Wright, T., Trefethen, L. N.: Large-scale computation of pseudospectra using ARPACK and Eigs. *SIAM J. Sci. Stat. Comput.* **23** (2001) 591:605 [202](#)
- [17] Dongarra, J., Duff, I., Sorensen, D., van der Vorst, H.: *Numerical Linear Algebra for High-Performance Computers*. SIAM (1998) [203](#)
- [18] Frommer, A., Glässner, U.: Restarted GMRES for shifted linear systems. *SIAM J. Sci. Comput.* **19** (1998) 15–26 [204](#)

Development and Tuning of Irregular Divide-and-Conquer Applications in DAMPVM/DAC*

Pawel Czarnul**

Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology, Poland
pczarnul@eti.pg.gda.pl

Abstract. This work presents implementations and tuning experiences with parallel irregular applications developed using the object oriented framework DAM-PVM/DAC. It is implemented on top of DAMPVM and provides automatic partitioning of irregular divide-and-conquer (DAC) applications at runtime and dynamic mapping to processors taking into account their speeds and even loads by other user processes. New implementations of parallel applications tuned for shortest execution time are investigated. They include $\alpha\beta$ search, recursive Fibonacci, $\binom{n}{k}$ and finding twin prime numbers in parallel. Various DAC parameters were tuned for specific applications including costs of computing vectors/-subtrees, maximum partitioning levels etc. Moreover, the overhead of DAMPVM/DAC compared to sequential implementations is shown including previously implemented adaptive quadrature integration and image recognition.

1 Introduction

DAMPVM/DAC ([1], [2]) is implemented on top of DAMPVM ([3]) and uses its dynamic features like process allocation and migration. It allows to use external load balancing algorithms (like the one proposed in [4]) thanks to a special communication protocol. Dynamic process partitioning is used to create at least as many processes as the number of available processors in order to keep all of them busy. Dynamic process migration is invoked before dynamic process partitioning if enough tasks are already available to balance the load or if it is necessary to move some tasks because some nodes have become overloaded by other users ([2]). Other similar systems either do not provide process migration or do not take multi-user environments into account. APERITIF (Automatic Parallelization of Divide and Conquer Algorithms, formerly APRIL, [5]) translates serial C programs to be run in parallel using PVM with no automatic compensation

* Work partially sponsored by the Polish National Grant KBN No. 8T11C 001 17

** Currently on leave at Electrical Engineering and Computer Science, University of Michigan, EECS, 1301 Beal Avenue, Ann Arbor, 48109 MI, U.S.A., pczarnul@eecs.umich.edu.

for irregularity. REAPAR (REcursive programs Automatically PARallelized, [6]) and Cilk ([7]) are thread-based approaches. Java is represented by ATLAS ([8]) and Satin ([9]) targeted for distributed memory utilizing work stealing. Frames ([10]) and Beeblebox ([11]) are other framework-based approaches.

With respect to mapping particular problems considered in this paper, [12] presents an excellent overview of parallel $\alpha\beta$ strategies. The $\alpha\beta$ algorithm implemented in this work is based on the Young Brothers Wait Concept (YBWC) explained in [13]. The leftmost branch of a (sub)tree needs to be evaluated first before its right brothers are spawned as separate processes and evaluated in parallel. [14], [13] describe a distributed chess program ZUGZWANG and various experiments with it. [12] and [15] (includes experiments on networks of workstations as opposed to the aforementioned works for shared memory systems) describe the APHID algorithm which partitions the tree among available nodes in a master-slave fashion and lets the slaves analyze their parts independently. The top part of the tree is assigned to the root processor which takes care of some fixed number of top levels of the tree and assigns higher levels (further from the root) to the slaves. [16] presents the ideas behind Chinook which won the World Championships in checkers. An algebraic model of divide-and-conquer is described in [17] while mapping strategies presented in [18] and [19].

2 DAMPVM/DAC Features and Tuning Mechanisms

The DAMPVM/DAC framework offers methods which enable the programmer to describe the functionality of a divide-and-conquer algorithm (Figure 1, [1]). The supported code is linked with a library of functions and executed with assistance of the runtime part of DAMPVM/DAC. Subtrees can be spawned as new processes, dynamically allocated and migrated to other nodes at runtime, if needed. By default, DAMPVM/DAC uses the abstraction of a vector which is passed and partitioned in recursive calls and represents data recursive calls operate on ([1]). If needed, it allows to use global variables and fetch data associated with the node of the tree currently being visited.

Asynchronous update messages. Although DAMPVM/DAC assumes that the order in which calls on a given level are executed does not change final results it may change the time required to analyze the tree as certain calls may use intermediate results produced by others. In $\alpha\beta$ search the values of α and β produced by previous calls can cut off large subtrees which results in shorter execution time ([15]). DAMPVM/DAC offers asynchronous messages which can be

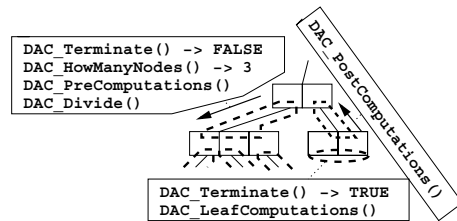


Fig. 1. DAMPVM/DAC Main Methods

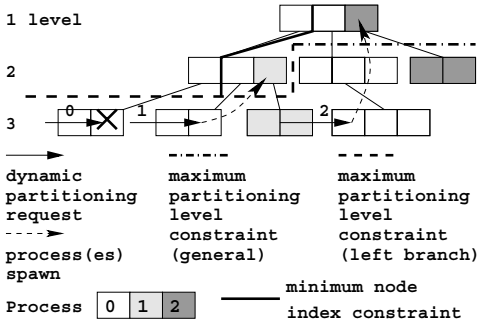


Fig. 2. Maximum Partitioning Levels

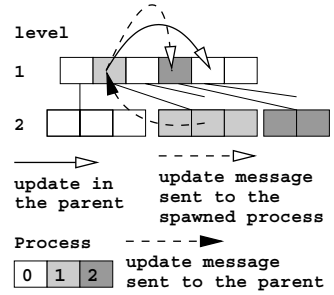


Fig. 3. Update Messages

sent to parallel processes spawned before by the same parent (Figure 3). Thanks to a handler function registered by the programmer, a process can intercept an asynchronous message, update its data and thus narrow the search space by cutting off subtrees.

Setting maximum levels and node indexes, partitioning processes of at least a given size. The implemented scheme enables to set maximum levels at which dynamic partitioning can be invoked (Figure 2). This prevents from spawning too short tasks with small subtrees. According to YBWC ([13]) in $\alpha\beta$ search subtrees should wait for the values of α and β from the leftmost subtree. The special partitioning level for the leftmost top level branch is usually set to higher values than the general one for the others. This enables to partition the leftmost top level branch while the others are waiting. In later stages of computations the maximum level is lower as all top level branches can be spawned. Moreover a minimum index of a node which may be partitioned at a given level can be set. Setting it to 2 prevents from spawning tasks at a certain level if the leftmost node at this level is still being executed (Figure 2). This allows to apply YBWC. Additionally, DAMPVM enables to partition only tasks larger than a predefined threshold. Tasks can report their sizes using method `DAC::DaC_VectorSize()` provided by the programmer. Although impossible to predict accurately for applications like the $\alpha\beta$ search this prevented small subtrees to be spawned as very short processes. This in turn prevented scenarios in which communication and synchronization costs were larger than parallelization gain.

3 Partitioning and Tuning Divide-and-Conquer Applications

The presented applications were implemented in DAMPVM/DAC, tuned as described below and executed on a LAN network on up to 16 SUN workstations. The results are shown in Figures 5 and 6 while the typical sizes of application

trees in Table 4. Maximum partitioning levels and asynchronous messages were required only by the most complex example which was the $\alpha\beta$ search. All could be optimized by providing accurate estimation of computational costs of subtrees assigned to processes which were used by DAMPVM/DAC schedulers. Largest processes were partitioned first.

$\alpha\beta$ Search Engines. As an example, a GoBang game was implemented in which two players put crosses and circles in turn on the board. The one who has first put its five pieces vertically, horizontally or diagonally wins. [15] explains three problems that prevent parallel implementations of the $\alpha\beta$ scheme from reaching high speed-ups: synchronization overhead/idle time, parallelization overhead due to communication between processors, process creation etc. and the search overhead mentioned above. The experiments (Figures 5 and 6) showed that the maximum partitioning level of 3 for the leftmost subtree and 2 for the others were optimal. Sets of 2 and 1 resulted in idle time for larger numbers of processors and higher levels in too small tasks being spawned. In the latter case large synchronization overhead greater than parallelization gain was observed. The minimum node index was set to 2 in order to force YBWC. The DAMPVM scheduler has been modified so that it sends dynamic partitioning requests to tasks larger than a certain threshold (60 for maximum depth 8 for position in Figure 4). The size of a task is determined using method `DAC::DaC_VectorSize()` which was set to $2^{max\ depth}$. This change has improved performance although is not accurate for irregular trees. The main problem is that the depth of the tree is low (Table 4) and dynamically spawned trees can be very small which causes large synchronization delays with the parent process. A simple move reordering ([20]) strategy was implemented:

1. Perform a quick shallow search sequentially (depth 6 for the position in Figure 4). No parallelization is done at this stage (DAC is disabled).
2. Put the best move at the beginning of the first level move list.
3. Execute the deeper search (of depth 8) in parallel.

[12] and [20] discuss iterative deepening and transposition tables. The above algorithm intentionally sacrifices some time by executing the shallow search. It compensates with shorter total execution time thanks to move reordering and using better α and β from the best move from the shallow search for cut-offs. [12] describes Scout search with zero-size $\alpha\beta$ windows in order to check whether a candidate move is better than the current best and its parallel version – Jamboree search. Move reordering can be applied to all moves at a given level.

Recursive Parallel Fibonacci and $\binom{n}{k}$. Although computing Fibonacci numbers recursively is inefficient it is regarded as a good benchmark for parallel divide-and-conquer systems since it is very fine-grained i.e. there are no computations in the leaves. Both applications are easily parallelized by DAMPVM/DAC due to large and reasonably balanced trees. Since the trees

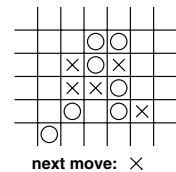


Fig. 4. GoBang Test Position

have large depths in the order of 1-40 (Table 4) even subtrees which are dynamically spawned at higher levels are still large. This implies that vector sizes could be set even to fixed values. Imbalance is detected by DAMPVM/DAC schedulers in a later stage of computations and partitioning is invoked again. Communication and synchronization costs are low compared to computational costs as opposed to unpredictable and often small sizes of subtrees in $\alpha\beta$ search. Figures 5 and 6 show results for $F(43)$ and $\binom{35}{13}$ respectively.

Finding Twin Primes. The goal is to find out how many twin prime numbers are in range $[a, b]$. The initial range $[a, b]$ ([1000432, 50000540] for the results in Figures 5 and 6) is divided into two parts which are then divided recursively until their lengths have reached a predefined number of elements (10000 in Figures 5 and 6). For large enough ranges of large numbers the tree is more balanced than for the irregular integration shown in [1] which results in a slightly better speed-up (Figure 5). The size of a vector is equal to the number of elements which may be a good estimate in this case. Notice that this problem was not formulated recursively. Alternatively, the initial vector could be divided statically into as many subvectors as the number of available processors. However, if a task is migrated to another machine when one is recaptured by the owner a considerable imbalance occurs since tasks can not be partitioned. The recursive formulation adds overhead due to recursive calls (Paragraph 4) but maintains the dynamic partitioning capability which enables to produce smaller tasks and achieve a smaller imbalance.

Adaptive integration and image recognition. These examples are analyzed in detail in [1] and [2] respectively. Comparing regular images (uniform patterns across the image, results for 4000x4000 pixels, 8-bit depth shown in Figures 5 and 6) to a smaller ROI (Region of Interest, 400x400, 8-bit depth in Figures 5 and 6) can be compared to the twin prime example with respect to the speed-up and the size of the tree (Table 4). The trees are reasonably large and balanced. Recognition of irregular images is similar to integration of irregular functions (the function takes different amounts of time for different subranges to be integrated, $\int_0^{7200\pi} \sin^2(x)dx + \int_{7200\pi}^{14400\pi} (x - 7200\pi)dx$ in Figures 5 and 6) due to unbalanced trees. In the latter case, the tree is deeper (Table 4) so it can be partitioned to smaller processes and mapping achieves a smaller imbalance. The size of a vector is set to the number of pixels in the assigned part of the image or the analyzed range respectively. [2] shows that migration is not profitable for large trees which can be divided into reasonably large subtrees by dynamic partitioning (like integration).

4 DAMPVM/DAC Overhead and Optimizations

DAMPVM/DAC introduces additional overhead to the recursive mechanism including:

1. tracking dynamic partitioning requests from DAMPVM schedulers,

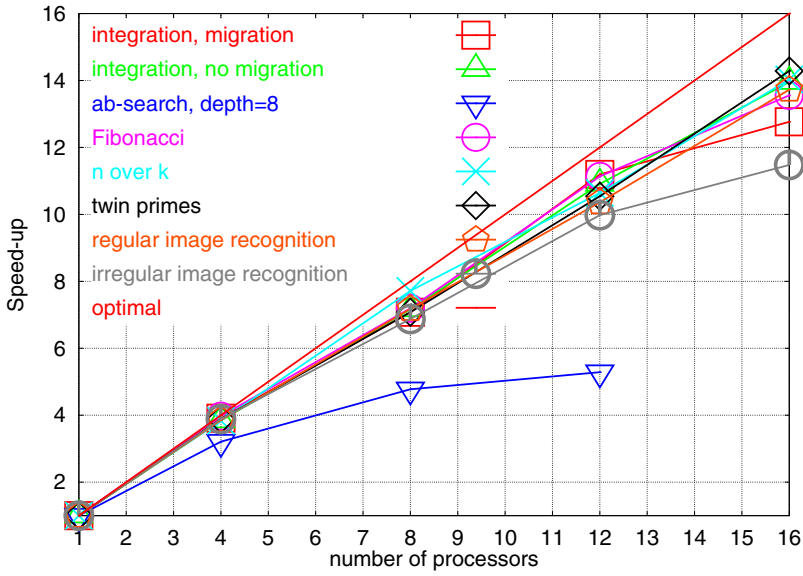


Fig. 5. Speed-up for the Implemented Divide-and-Conquer Applications

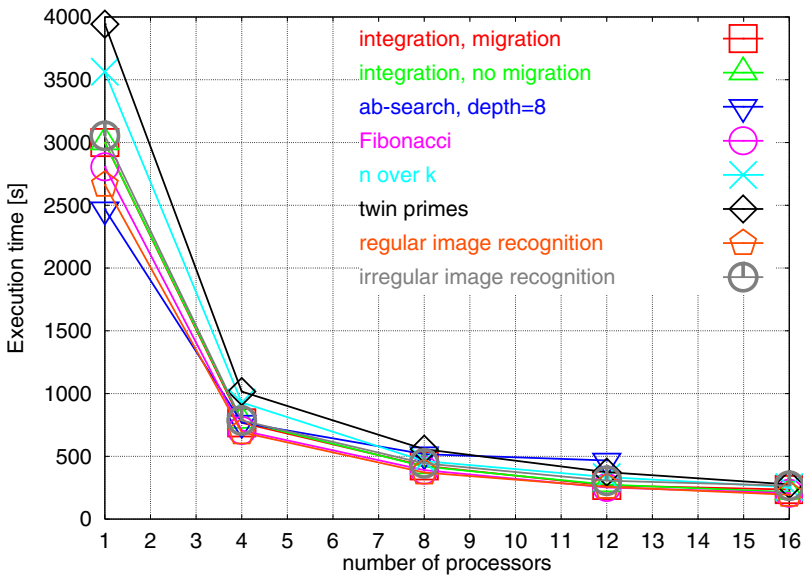


Fig. 6. Execution Time of the Implemented Divide-and-Conquer Applications

Table 1. DAMPVM/DAC Overhead and Typical Trees Tested

| Applica- tion | Problem Size | Exec. time without DAC (t_s) | Exec. time with DAC (t_1) | Ratio $\frac{t_1}{t_s}$ | Node de- gree | Tree depths tested |
|---------------------------|--|---|--|----------------------------|---------------------|--------------------------|
| Adaptive Integration | $\int_0^{240\pi} \sin^2(x) dx$ | 111.6 | 115.8 | 1.04 | 2 | \approx 1-25 |
| $\alpha\beta$ Search | shallow search depth=4, actual search depth=7 | 52 | 52 | 1 | \approx 5-30 | < 10 |
| Fibonacci | F(37) | 70.5 | 103.2 | 1.46 | 2 | \approx 1-40 |
| $\binom{n}{k}$ | $\binom{29}{13}$ | 100.5 | 139.2 | 1.39 | 2 | \approx 1-30 |
| Twin Primes | range [1000432, 5000540] | 81.1 | 84.1 | 1.04 | 2 | \approx 10 |
| Image Recogni- tion | image: 1000x1000, ROI: 40x40, leaf limit: 60 pixels | 112.8 | 118.9 | 1.05 | 4 | \approx 1-5 |

2. storing: the pointers, the number of vectors and indexes of analyzed nodes at various levels, highest dynamic partitioning levels for next DAC invocations,
3. setting process sizes at runtime: after new tasks are spawned, before and after data is gathered from processes spawned at the currently visited level.

The overhead can be evaluated by comparing the execution time of an application using DAMPVM/DAC and the best possible static implementation, both executed on one processor. All the presented examples have been implemented without DAMPVM/DAC as well. The results are presented in Table 4. The worst (highest) $\frac{t_1}{t_s}$ ratios were obtained for the Fibonacci and $\binom{n}{k}$ examples. This is because there are practically no computations performed in the leaves, the trees are reasonably large and the overhead described above is considerable compared to real divide-and-conquer operations executed at each level of recursion. [9] compares the overhead of a similar Satin system where invocation records need to be registered to other implementations like Cilk ([7]) or ATLAS ([8]). Thresholding techniques similar to those described below are mentioned. DAMPVM/DAC shows the overhead very similar or lower than Satin.

Migration in DAMPVM is based on the PVM 3.4 message handlers ([21]). Signals are caught as soon as any PVM function is called ([22]). Thus the current DAC implementation contains calls to `pvm_nrecv()` from an unexisting source waiting for an unexisting message. It was shown that no visible overhead is introduced when this function is called up to level 15 for narrow but very deep trees like the Fibonacci example and up to level 6 for wide and shallow trees like the GoBang example. These values do not restrict the dynamic partitioning capability since the subtrees below these levels are reasonably short and trees are always partitioned from the root so they do not reach these limits in practice.

5 Conclusions and Future Work

It was shown that the Fibonacci, $\binom{n}{k}$, twin prime examples and irregular applications like integration of irregular functions, recognition of irregular images benefit from efficient automatic partitioning in DAMPVM/DAC due to reasonably large trees and the ease of evaluating the computational costs of subtrees. On the other hand, in the $\alpha\beta$ search there is a trade-off between the reduction of execution time due to parallelization and the costs: both the search (reduced by asynchronous update messages) and synchronization/communication overhead. Experimentally derived maximum levels and the minimum size of a process for partitioning try to prevent synchronization costs higher than the parallelization gain. Future work will be focused on merging remote clusters running DAMPVM/DAC using tools like PHP or Java servlets.

References

- [1] Czarnul, P., Tomko, K., Krawczyk, H.: Dynamic Partitioning of the Divide-and-Conquer Scheme with Migration in PVM Environment. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2131 in Lecture Notes in Computer Science, Springer-Verlag (2001) 174–182 8th European PVM/MPI Users' Group Meeting, Santorini/Thera, Greece, September 23–26, 2001, Proceedings 208, 209, 212
- [2] Czarnul, P.: Dynamic Process Partitioning and Migration for Irregular Applications, Warsaw, Poland (2002) *accepted for International Conference on Parallel Computing in Electrical Engineering PARELEC'2002*, <http://www.parelelec.org> 208, 212
- [3] Czarnul, P., Krawczyk, H.: Dynamic Assignment with Process Migration in Distributed Environments. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 1697 in Lecture Notes in Computer Science (1999) 509–516 208
- [4] Czarnul, P., Tomko, K., Krawczyk, H.: A Heuristic Dynamic Load Balancing Algorithm for Meshes. In: Parallel and Distributed Computing and Systems, Proceedings of the IASTED International Conference, Anaheim, CA, USA (2001) 166–171 208
- [5] Erlebach, T.: APRIL 1.0 User Manual, Automatic Parallelization of Divide and Conquer Algorithms. Technische Universitat Munchen, Germany, <http://wwwmayr.informatik.tu-muenchen.de/personen/erlebach/aperitif.html>. (1995) 208
- [6] Prechelt, L., Hänßgen, S.: Efficient parallel execution of irregular recursive programs. IEEE Transactions on Parallel and Distributed Systems 13 (2002) 209
- [7] R. D.Blumofe, C. F.Joerg, B. C.Kuszmaul, C. E.Leiserson, K. H.Randall, Y.Zhou: Cilk: An efficient multithreaded runtime system. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (1995) 207–216 209, 214
- [8] J. Baldeschwieler, R. Blumofe, E. Brewer: ATLAS: An Infrastructure for Global Computing. In: Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications. (1996) 209, 214

- [9] van Nieuwpoort, R. V., Kielmann, T., Bal, H. E.: Satin: Efficient Parallel Divide-and-Conquer in Java. In: Euro-Par 2000 Parallel Processing, Proceedings of the 6th International Euro-Par Conference. Number 1900 in LNCS (2000) 690–699 [209](#), [214](#)
- [10] I. Silvestre, J. P., Romke, T.: Programming Frames for the efficient use of parallel systems. Technical Report TR-183-97, Paderborn Center for Parallel Computing (1997) [209](#)
- [11] Piper, A., Prager, R.: Generalized Parallel Programming with Divide-and-Conquer: The Beeblebrox System. Technical Report CUED/F-INFENG/TR132, Cambridge University Engineering Department (1993) <ftp://svr-ftp.eng.cam.ac.uk/pub/reports/piper-tr132.ps.Z> [209](#)
- [12] Brockington, M.: Asynchronous Parallel Game-Tree Search. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Canada (1998) www.cs.ualberta.ca/~games/articles/mgb.thesis.ps.gz [209](#), [211](#)
- [13] Feldmann, R., Mysliwicz, P., Monien, B.: Studying Overheads in Massively Parallel MIN/MAX-Tree Evaluation. In: Proc. of the 6th ACM Symposium on Parallel Algorithms and Architectures (SPAA '94). (1994) 94–103 http://www.uni-paderborn.de/fachbereich/AG/monien/PUBLICATIONS/POSTSCRIPTS/FMM-Studying-Overheads_1994.ps.Z [209](#), [210](#)
- [14] Feldmann, R., Mysliwicz, P., Monien, B.: Experiments with a fully distributed chess program system. Heuristic Programming in Artificial Intelligence 3 (1992) 72–87 Ellis Horwood Publishers, <ftp://ftp.uni-paderborn.de/doc/techreports/Informatik/tr-ri-94-139.ps.Z> [209](#)
- [15] Brockington, M. G., Schaeffer, J.: The APHID Parallel alpha-beta Search Algorithm. In: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96). (1996) [209](#), [211](#)
- [16] Schaeffer, J.: Search Ideas in Chinook. Games in AI Research (2000) 19–30 <http://www.cs.ualberta.ca/~jonathan/Papers/Papers/searchideas.ps> [209](#)
- [17] Mou, Z. G., Hudak, P.: An algebraic model for divide-and-conquer and its parallelism. The Journal of Supercomputing **2** (1988) 257–278 [209](#)
- [18] Lo, V., Rajopadhye, S., Telle, J., Zhong, X.: Parallel Divide and Conquer on Meshes. IEEE Transactions on Parallel and Distributed Systems **7** (1996) 1049–1057 [209](#)
- [19] Wu, I.: Efficient parallel divide-and-conquer for a class of interconnection topologies. In: Proceedings of the 2nd International Symposium on Algorithms. Number 557 in Lecture Notes in Computer Science, Taipei, Republic of China, Springer-Verlag (1991) 229–240 [209](#)
- [20] Verhelst, P.: Chess Tree Search. (<http://www.xs4all.nl/~verhelst/chess/search.html>) [211](#)
- [21] Geist, A.: Advanced Tutorial on PVM 3.4 New Features and Capabilities (1997) Advanced Tutorial on PVM 3.4 Presented at EuroPVM-MPI'97, <http://www.csm.ornl.gov/pvm/EuroPVM97/> [214](#)
- [22] Oak Ridge National Laboratory, U. S. A.: (PVM Documentation: Listing of new features found in PVM 3.4 and past PVM versions) <http://www.epm.ornl.gov/pvm/changes.html#pvm3.4.0> [214](#)

Observations on Parallel Computation of Transitive and Max-Closure Problems^{*}

Aris Pagourtzis^{1,2**}, Igor Potapov³, and Wojciech Rytter^{3,4}

¹ Institute of Theoretical Computer Science, ETH Zürich

`pagour@inf.ethz.ch`

² Department of Elec. Comp. Eng., National Technical University of Athens, Greece

³ Department of Computer Science, University of Liverpool, Chadwick Building

Peach St, Liverpool L69 7ZF, U.K.

`{igor,rytter}@csc.liv.ac.uk`

⁴ Institute of Informatics, Warsaw University, Poland

Abstract. We present new approaches to the parallel computation of a class of problems related to the GENERIC TRANSITIVE CLOSURE problem (TC, in short). We identify the main ingredient of the TC problem called the MAX-CLOSURE Problem and concentrate on parallel computation of this subproblem, also we show how to reduce TC to matrix multiplication once the max-closure is computed. We present a new variation of the Warshall algorithm for MAX-CLOSURE, both in fine-grained and coarse-grained forms; the coarse-grained version, appropriate for parallel implementation in PVM is especially designed so as to consist of highly parallelisable submatrix multiplications. We used existing, especially efficient PVM subroutines to realise our new MAX-CLOSURE and TC algorithms; the experimental results show that the new algorithms achieve considerable improvement compared to previously known ones.

1 Introduction

An important computational problem with numerous applications in many areas of computer science is the *generic transitive closure* problem (TC, in short). It can be defined using an abstract notion of a *semiring* \mathcal{S} with operations \oplus and \otimes , see [1] for formal definitions. Starting from a graph with weights (elements of the semiring \mathcal{S}) on its edges, the goal is to sum the weights of the paths from a given vertex i to a given vertex j using the generalised summation operation \oplus ; the weight of each path is the product of the weights of its edges in terms of the generalised multiplication operation \otimes . In the case of the TC problem we sum only simple paths, i.e. paths in which no vertex appears twice.

GENERIC TRANSITIVE CLOSURE PROBLEM (TC)

Input: a matrix A with elements from a semiring \mathcal{S} .

^{*} Work partially supported by GR/N09855 EPSRC grant.

^{**} Part of this work was done while this author was with the Department of Computer Science, University of Liverpool.

Output: the matrix A^* , $A^*(i, j)$ is the sum of all *simple* paths from i to j . We study the problem restricted in *simple* semirings (first defined as Q -semirings in [10]). Properties of simple semirings imply that, to compute TC it suffices to sum all simple paths *at least once*: adding a path more than once or adding a non-simple path does not make any difference.¹ Two straightforward applications of the TC problem are the *boolean closure* of a 0-1 matrix (where \oplus and \otimes are the boolean *OR* and boolean *AND* operations respectively)—equivalent to computing the *transitive closure* of a directed graph—and *all-pairs shortest paths* computation in graphs (where \oplus and \otimes are the *MIN* and $+$ operations). Less clear-cut is the computation of *minimum spanning trees* [8]: if we consider \oplus and \otimes as *MIN* and *MAX*, then the minimum spanning tree consists of all edges (i, j) such that $A(i, j) = A^*(i, j)$, assuming that all initial weights $A(i, j)$ are distinct.

The best known sequential algorithm for the TC problem is the *Warshall* algorithm [1] which works in cubic time. This algorithm can be implemented in message passing systems, such as PVM, by using its coarse-grained version, called *Warshall Coarse-Grained*. A second coarse-grained algorithm, working in two passes has been proposed in [2]; this algorithm uses some ideas from the Guibas-Kung-Thompson systolic algorithm [4], hence it was called *GKT-Coarse-Grained (GKT-CG)* algorithm. We observe that the GKT-CG algorithm works in two passes, computing what we call *max-closure* of the matrix (see section 2 for definition) in its first pass (GKT-1P for short) and the transitive closure in its second pass (GKT-2P).

In this paper, we identify max-closure as an important ingredient of transitive closure which is interesting *per se*. For example, in graphs with appropriate ordering the max-closure coincides with transitive closure. Moreover, we show that after computing the max-closure, one matrix multiplication suffices to give the transitive closure. We also show that max-closure itself can be computed by a sequence of submatrix multiplications. All (sub)matrix multiplications can be computed very efficiently in parallel by existing PVM subroutines. We present a new coarse-grained parallel algorithm for max-closure computation which can be seen as a *Partial Coarse-Grained Warshall* algorithm; we compare its performance to the performance of GKT-1P. We show experimentally that our new algorithm computes the max-closure faster and therefore, combined with GKT-2P gives a faster algorithm for computing the Transitive Closure of a graph.

2 Notation

Given a graph $G = (V, E)$ and two nodes $i, j \in V$ a *path* p from i to j is a sequence $\langle i, k_1, \dots, k_l, j \rangle$ such that there is an edge between any two consecutive nodes. If $l > 0$ we say that path p has internal nodes; each node k_i is an internal node

¹ Actually, these properties imply that in simple semirings the sum of *all* paths equals the sum of all simple paths. Hence, the algorithm presented in this paper solves in fact an even more general problem, the *Algebraic Path Problem (APP)*, in the special but important case of simple semirings; see [3] for a nice survey on APP.

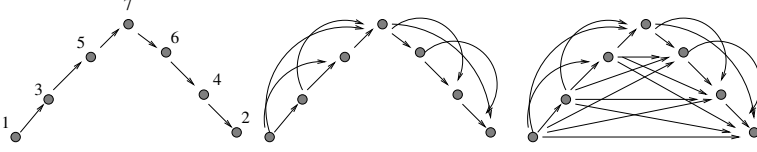


Fig. 1. An example graph, its max-closure and its transitive closure

of p . Node k_1 is then the *first internal node* and k_l is the *last internal node*. If G is edge-weighted then the weight of p is the generalized product of the weights of the edges that connect consecutive nodes in p . Let $i \xrightarrow{S} j$ denote a path from i to j with internal nodes that belong to the set S . Abbreviate $i \xrightarrow{\{1, \dots, k-1\}} j$ by $i \xrightarrow{\leq k} j$. In other words, $i \xrightarrow{\leq k} j$ denotes any path from i to j with internal nodes smaller than k . Similarly, we will abbreviate $i \xrightarrow{\{1, \dots, k\}} j$ by $i \xrightarrow{\leq k} j$. In the sequel we will usually start with a matrix A that initially contains information about edges of the graph. During the computation, A is updated by adding to each $A(i, j)$ the weight of paths from i to j . Let p be a path from i to j . We will use the expression to *process* p to mean “to calculate the weight of p and add this value to $A(i, j)$ ”.

A *max-path* from i to j is a path whose intermediate nodes have numbers smaller than $\max(i, j)$. Denote by A^{max} the *max-closure* of A , i.e. for each i, j $A^{max}(i, j)$ is the sum of all simple max-paths from i to j . The goal of a max-closure algorithm is thus to process all *simple* paths $i \xrightarrow{\leq \max(i, j)} j$ for all pairs of nodes (i, j) , while the goal of a TC algorithm is to process all simple paths between i and j for all pairs (i, j) . (See Figure 1 for a comparison of max-closure and transitive closure.)

3 A Fine Grained Parallel Algorithm

We will show in this section that transitive closure is closely related to max-closure and matrix multiplication. Matrix multiplication can be done in parallel in a very simple way:

for all i, j parallel do $A(i, j) = \sum_{k=1}^n A(i, k) \otimes A(k, j)$

We define an elementary fine-grained operation:

Operation (i, k, j) : $A(i, j) := A(i, j) \oplus A(i, k) \otimes A(k, j)$;

Matrix multiplication requires a cubic number of elementary operations. It is known that transitive can be also computed by a cubic number of elementary operations. We will show that transitive closure of a $n \times n$ matrix A can be computed by first computing its max-closure A^{max} , using slightly less than $2n^3/3$ elementary operations, and then performing a multiplication of triangular matrices, requiring slightly less than $n^3/3$ elementary operations. For a matrix A , denote by A^\triangleleft the upper triangular matrix that is defined as follows: $A^\triangleleft(i, j) = A(i, j)$ if

$i \leq j$, $A^\triangleright(i, j) = 0$ otherwise. Let us also define the lower triangular matrix A^\triangleright similarly: $A^\triangleright(i, j) = A(i, j)$ if $i \geq j$, $A^\triangleright(i, j) = 0$ otherwise.

Lemma 1. (*Reduction: TC \Rightarrow MM*) *If $A = A^{max}$, then $A^* = A \oplus (A^\triangleleft \otimes A^\triangleright)$.*

Proof. For any i, j , consider any simple path p from i to j . Let k be the greatest internal node. If $k < \max(i, j)$ then this is a *max-path* from i to j , and therefore $A (=A^{max})$ already contains the path's value. Let us now suppose that $k > \max(i, j)$. Then p consists of two subpaths: $i \xrightarrow{\leq k} k$ and $k \xrightarrow{\leq k} j$. Both subpaths are *max-paths*, hence A already contains their values in $A(i, k)$ and $A(k, j)$ respectively (since $A = A^{max}$). We only need to show that *Operation*(i, k, j) is one of the elementary operations performed by the operation $A^* = A \oplus (A^\triangleleft \otimes A^\triangleright)$. This is true since $k > \max(i, j)$ and hence $A(i, k)$ belongs to the non-zero part of A^\triangleleft and $A(k, j)$ belongs to the non-zero part of A^\triangleright .

The Algorithm Max-to-Transitive shown below performs the operation $A = A \oplus (A^\triangleleft \otimes A^\triangleright)$. Then, by Lemma 1, the following theorem holds.

Theorem 1. *Algorithm Max-to-Transitive computes the Generic Transitive Closure of a matrix A .*

Algorithm Max-to-Transitive

Input: matrix A , such that $A^{max} = A$

Output: transitive closure of A

$\{A := A \oplus (A^\triangleleft \otimes A^\triangleright)\}$

for all $k \leq n$ **parallel do**

for all i, j , $\max(i, j) < k$, $i \neq j$
parallel do *Operation*(i, k, j)

The classical algorithm for computing the transitive closure is the Warshall algorithm which performs slightly less than n^3 elementary operations. We present its parallel *partial* version which computes only A^{max} using exactly a fraction 2/3 of the same elementary operations that Warshall algorithm performs. By making use of this algorithm, one can also compute TC by performing the remaining 1/3-fraction of operations *completely in parallel* as shown in the previous section.

Algorithm Max-Closure {Partial-Warshall}

for $k := 1$ **to** n **do**

for all $1 \leq i, j \leq n$, $\max(i, j) > k \neq \min(i, j)$, $i \neq j$
parallel do *Operation*(i, k, j)

Theorem 2. *Algorithm Max-Closure computes the max-closure of the input matrix A .*

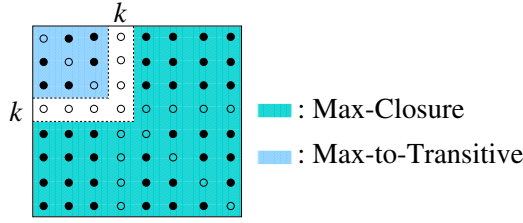


Fig. 2. Black nodes in lower-right (resp. upper-left) region represent elements $A(i, j)$ for which $Operation(i, j, k)$ is performed during step k of Max-Closure Algorithm (resp. Max-to-Transitive Algorithm). Note that in Max-to-Transitive Algorithm all steps, $1 < k \leq n$, are performed in parallel

Proof. It suffices to show that the following holds for all $k \in 1, \dots, n$:

Claim: After step k of the algorithm all simple paths $i \xrightarrow{\leq k} j$ are processed for all i, j such that $\max(i, j) > k$. We will prove that by induction on k . The claim is true for $k = 1$, since all paths $i \xrightarrow{\leq 1} j$ are either original edges or consist of two original edges $i \rightarrow 1$ and $1 \rightarrow j$ (hence, in this case $i \neq 1 \neq j$). In the latter case the path is processed by $Operation(i, 1, j)$, which is performed in step 1 for all $i, j > 1$. Assume now that the claim holds for step $k - 1$. Consider any simple path p of type $i \xrightarrow{\leq k} j$ with $\max(i, j) > k$. If the greatest internal node is at most $k - 1$, then the claim holds by our assumption. If the greatest internal node is k then p consists of two subpaths: $p_1 : i \xrightarrow{\leq k-1} k$ and $p_2 : k \xrightarrow{\leq k} j$. Since $\max(i, k), \max(k, j) > k - 1$ both paths are processed after step $k - 1$, by our assumption. Since $\max(i, j) > k$ and $i \neq j$ $Operation(i, k, j)$ is performed during step k . Thus, p is correctly processed.

In Warshall algorithm, the number of elementary operations $Operation(i, k, j)$ equals the number of (i, k, j) permutations $n(n-1)(n-2)$, since we only need to perform $Operation(i, k, j)$ for pairwise different i, k, j . Our algorithm for computing transitive closure performs the same number of operations—but in two phases as shown in Figure 2. The first phase corresponds to the max-closure computation (Algorithm Max-Closure) and the second corresponds to the transitive closure computation by multiplication of the triangular parts of the max-closure matrix (Algorithm Max-to-Transitive). The key-point is that in the second phase *all* operations can be performed in parallel, so it is important to see how many they are. Denote by $oper_1(n)$ and $oper_2(n)$ the number of elementary operations in phase 1 and phase 2, respectively, and by $oper(n)$ the total number of elementary operations performed in both phases.

The following theorem shows that $1/3$ of the operations are performed in the second phase, i.e. completely in parallel.

Theorem 3. $oper_2(n) = \frac{1}{3}oper(n).$

Proof. Figure 2 shows exactly which operations $Operation(i, k, j)$ are performed for a specific k during the first and the second phase. It is not hard to see that the total number of operations is the same as in Warshall algorithm: $oper(n) = n(n-1)(n-2)$. In phase 2, for any k the number of elementary operations is $(k-1)(k-2)$. Altogether we have:

$$oper_2(n) = \sum_{k=1}^n (k^2 - 3k + 2) = \frac{n(2n+1)(n+1)}{6} - 3\frac{n(n+1)}{2} + 2n = \frac{1}{3}oper(n)$$

4 A Coarse-Grained Max-Closure Algorithm

We will now give a coarse-grained version of the Max-Closure algorithm, namely the CG-Max-Closure Algorithm. In coarse-grained computations it is necessary to split the initial matrix A into several blocks—usually the number of blocks is p , where p is the number of available processors. Let $N = \lfloor \sqrt{p} \rfloor$. We denote these blocks by $\tilde{X}_{I,J}$, $1 \leq I, J \leq N$ (by convention we use capital letters to denote indices of blocks). A corresponding partition of the set of nodes V into N subsets is implicit; each block $\tilde{X}_{I,J}$ contains information for paths between all pairs (i, j) , $i \in V_I, j \in V_J$. For each node i let $b(i)$ be the index of the subset of V that contains i . For brevity, we will use I for $b(i)$, J for $b(j)$, and so on, whenever this does not cause any confusion. We define the corresponding elementary operation for blocks:

$$Block-Operation(I, K, J): \tilde{X}(I, J) := \tilde{X}(I, J) \oplus \tilde{X}(I, K) \otimes \tilde{X}(K, J);$$

One might try a naive transformation of the fine-grained Partial Warshall algorithm, described in Section 3, by replacing elementary operations with block-elementary operations, and indices of elements with indices of blocks. Of course, operations in which not all I, J, K are different would now be meaningful. Such an approach does not work correctly as can be shown by simple examples (omitted here due to space limitations). Additional computations are required, related to the diagonal blocks $\tilde{X}_{K,K}$. The correct algorithm is shown below. A theorem similar to Thm. 2 can be stated and proved for algorithm CG-MC; this is also omitted for shortness.

Algorithm CG-Max-Closure {Partial Blocks-Warshall}

```

for  $K := 1$  to  $N$  do
   $\tilde{X}(K, K) := \tilde{X}(K, K)^*$ 
  for all  $1 \leq J \leq N$ , and  $1 \leq I \leq N, I \neq K \neq J$ 
    parallel do  $Block-Operation(K, K, J)$ 
      and  $Block-Operation(I, K, K)$ 
  for all  $1 \leq I, J \leq N, \max(I, J) > K \neq \min(I, J)$ 
    parallel do  $Block-Operation(I, K, J)$ 

```

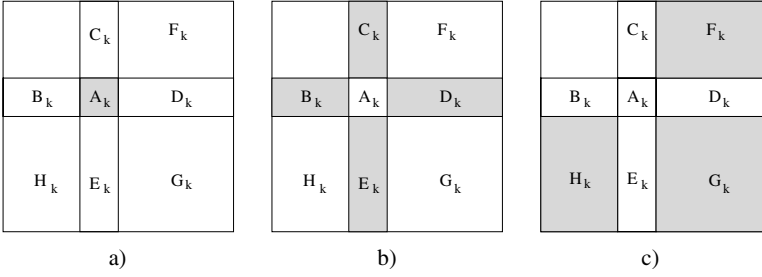


Fig. 3. Representation of the Coarse-Grained Max-Closure (CG-MC) Algorithm. Snapshots (a), (b) and (c) show the sequence of regions that are processed during step K

5 Implementation and Comparison of the Algorithms

We now show the implementation of the algorithm CG-MC in terms of multiplication of submatrices. Each submatrix multiplication (usually) involves several block multiplications which are all performed in parallel. This way, we can make use of existing efficient PVM (or other message passing) procedures for submatrix multiplication. In what follows, the statements connected by the symbol ‘||’ are independent of each other and are performed in parallel.

Coarse-Grained Max-Closure Algorithm: Implementation

```

for  $K := 1$  to  $N$  do    /* the submatrices are as in Figure 3 */
   $A_K := A_K^*$ ;
   $B_K \oplus = A_K \otimes B_K$  ||  $C_K \oplus = C_K \otimes A_K$  ||  $D_K \oplus = A_K \otimes D_K$  ||  $E_K \oplus = E_K \otimes A_K$ ;
   $F_K \oplus = C_K \otimes D_K$  ||  $G_K \oplus = E_K \otimes D_K$  ||  $H_K \oplus = E_K \otimes B_K$ ;

```

We compare experimentally the practical behavior of two algorithms for max-closure computation: the first pass of the coarse-grained version of the Guibas-Kung-Thompson algorithm (GKT-1P) [2] and our Coarse-Grained Max-Closure Algorithm (CG-MC). All experiments were performed on a cluster of Hewlett Packard 720 Unix workstations networked via a switched 10Mb Ethernet, using PVM version 3.4.6 as message passing system.

The real (wall-clock) running time of the two algorithms for three different size matrices are presented in Figure 4. The results show that our new Max-Closure algorithm consistently outperforms the first pass of the GKT Algorithm. Both algorithms may use the same Max-to-Transitive algorithm as a second pass to compute the transitive closure. It turns out that improving computation time for max-closure by 20-30% gives around 16-24% improvement of overall computation of TC problem. We observe that for more than 16 processors there is an increase of the overall time (or a sudden fall of the time decrease rate). We give two possible reasons: a) While computation time decreases when adding processes the communication time increases. Since the latter is usually much

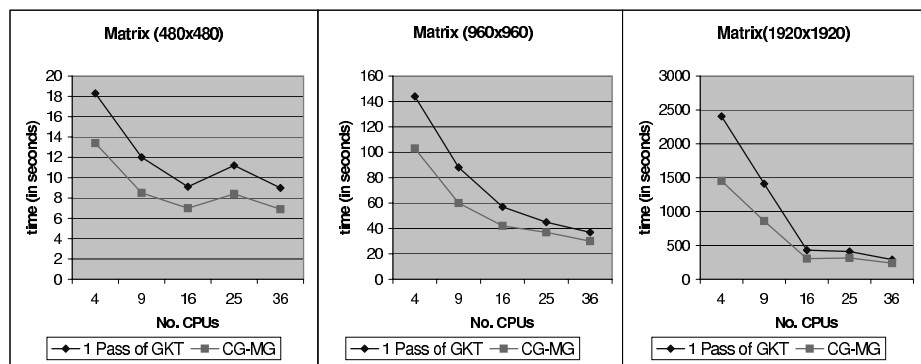


Fig. 4. Experimental results for parallel computation of Max-Closure. Comparison of the first phase of the algorithm of [2] (called GKT-1P), which computed Max-Closure, and of our coarse-graine algorithm (called CG-MC)

more costly there is an “ideal” number of processes which exhibits the minimum overall time. b) All experiments were carried out on a cluster of 20 workstations. This means that when more than 20 worker-processes were employed, some processors were running more than one worker-process. However, due to low communication cost and to time-slicing used by most operating systems we may have some further decrease by adding even more processes; this explains the two minimal points in the 480×480 curve (Figure 4).

The major advantage of the GC-MC algorithm, revealed by the experiments, is the reduction of communication cost at the expense of small computation cost. This fact makes CG-MC (as well as the corresponding TC algorithm) particularly useful for systems with slow communication, e.g. bus networks; therefore, we believe that our algorithms will prove to be very useful, especially for parallel distributed computing over wide area networks and, ultimately, over the Internet.

References

- [1] A. Aho, J. Hopcroft, J. Ullman, The design and analysis of computer algorithms, Addison-Wesley (1974). 217, 218
- [2] K. Chan, A. Gibbons, M. Pias, W. Rytter, On the PVM Computations of Transitive Closure and Algebraic Path Problems, in Proc. EuroPVM/MPI’98, 1998. 218, 223, 224
- [3] E. Fink, A Survey of Sequential and Systolic Algorithms for the Algebraic Path Problem, Technical Report CS-92-37, Dept. of CS, University of Waterloo, 1992. 218
- [4] L.Guibas, H.-T.Kung, C.Thompson, Direct VLSI implementation of combinatorial algorithms, Caltech Conf. on VLSI (1979). 218
- [5] H.-T.Kung and Jaspal Subhlok, A new approach for automatic parallelization of blocked linear algebra computations, 122–129, 1992.

- [6] Sun-Yuan Kung, Sheng-Chun Lo, Paul. S. Lewis, Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems, *IEEE Transactions on Computers*, C-36, 5, 603–614, 1987.
- [7] Hans-Werner Lang, Transitive Closure on the Instrucion Systolic Array, 295–304, *Proc. Int. Conf. on Systolic Arrays*, San Diego, 1991.
- [8] B. Maggs, S. Plotkin, Minimum cost spanning trees as a path finding problem, *IPL* 26 (1987) 191–293. 218
- [9] Gunter Rote, A systolic array algorithm for the algebraic path problem, *Computing*, 34, 3, 191–219, 1985.
- [10] M. Yoeli, A note on a generalization of Boolean matrix theory, *American Mathematical Monthly*, 68 (1961), 552–557. 218

Evaluation of a Nearest-Neighbor Load Balancing Strategy for Parallel Molecular Simulations in MPI Environment^{*}

Angela Di Serio and María B. Ibáñez

Departamento de Computación y Tecnología de la Información
Universidad Simón Bolívar
Apartado 89000, Caracas 1080-A, Venezuela
{adiserio,ibanez}@ldc.usb.ve

Abstract. We evaluate the effectiveness of an iterative nearest-neighbor load balancing algorithm: Generalized Dimension Exchange (G.D.E.) method over Molecular Dynamics Simulations.

Due to the overhead of the method, the simulation performance improves when the system tolerates some degree of unbalance. We present a method to calibrate the best tolerance level for a simulation running on a homogeneous platform using MPI.

1 Introduction

Molecular dynamics (MD) is a powerful tool used to study the properties of molecular systems, and their dynamic interactions. A large number of measurable or immeasurable static or dynamical properties can be obtained [5, 6, 9].

A typical MD simulation may run for days. At each timestep of the simulation, interactions among the molecules are computed using a predefined potential such as Lennard-Jones, then the coordinates and the velocity of each molecule are updated by solving the motion equations. This procedure is repeated until the end of the simulation. The molecules interactions are independent and can be done in parallel.

For parallel MD simulations, three decomposition schemes have been proposed, namely particle decomposition, force decomposition, and spatial decomposition [10]. In the spatial decomposition scheme, spatial domain of molecules is geometrically decomposed into disjoint parts, each of which is assigned to a different processor. Initially, all the processors have almost the same amount of molecules. Since molecules dynamically change their positions, the workload of the processors changes over the course of a computation, and cannot be estimated beforehand. Parallelization becomes inefficient after some time. In order to minimize the execution time, it is necessary to equally distribute the workload over all the processors in the system. One of the most popular load balancing

^{*} This work has been supported by the Project Agenda Petróleo (CONICIT) N. 97003530

methods used is the recursive geometric bisection [4, 7]. However, this kind of method needs global synchronization and intensive data transfer among the processors that results in poor scalability [11]. Centralized schemes have also been used [8] but they have a high communication overhead and poor scalability. To the best of our knowledge, very limited studies on dynamic load balancing over MD simulations have been carried out.

In recent years, dynamic load balancing algorithms have been developed for an important class of scientific applications [14, 11, 13]. There is growing interest in evaluating the behavior of these techniques under different network topologies [2, 1, 12].

Our aim is to evaluate the behavior of a dynamic distributed load balancing method: G.D.E. (Generalized Dimension Exchange [14]) applied to MD simulations. The simulations are carried out over a homogeneous Pentium cluster with a Myrinet interconnection using MPI as message-passing environment. In this work, we analyze the effectiveness of the G.D.E. method in MD simulations, and we present a simple and inexpensive heuristic to decide when to initiate the G.D.E. algorithm.

This paper is organized as follows. Section 2 briefly describes parallel molecular dynamics (MD) simulation on distributed memory machines. The G.D.E. method is outlined in Section 3. In Section 4, we present a trigger condition for load balancing in MD simulations that can be used on any homogeneous platform. In Section 5, we determine whether the load balancing strategy is suitable for MD applications and we show the effectiveness of the analytical model proposed. The conclusions are presented in Section 6.

2 Parallel Molecular Dynamics Simulations

Molecular dynamics simulations use classical models of motion and interaction. Several techniques have been developed for parallelizing short-range MD simulations; one of them is the spatial decomposition approach in which each process is responsible for simulating a portion of the system domain. In the integration of the equations of motion, the n molecules can move independently, leaving the space of the processor and entering the space of another one. Thus, the parallelization becomes inefficient after some time and load balancing must then be applied to increase efficiency.

An iteration step of the Plimpton code [10] uses an $n \times n$ matrix F (the F_{ij} element represents the force on molecule i due to molecule j) and X , a vector of length n which stores the position of each molecule. Each iteration consists of the following steps:

Typically, the program spends around 80% of the time in step one of the algorithm, 15% in step two, and 5% doing the communication work needed by steps one and three (see Fig. 1). Communication is done between neighbor processors to interchange molecules and it is implemented in a MPI environment.

- (1) Construct neighbor lists of non-zero interactions
in F every 20 steps
- (2) Compute elements of F
- (3) Update molecule positions in X using F

Fig. 1. General program structure

3 Load Balancing Operation

Usually, a load balancing operation consists of four steps:

- 1. Information Exchange** The processors collect information about the load of each node of the system.
- 2. Initiation Rule** The processors decide whether the load balancing should be initiated.
- 3. Load Transfer Calculation** The processor computes the amount of workload that the processor expects to give or receive from its direct neighbors.
- 4. Load Balance Operation** The workload assigned to the processors is modified. This step involves migration of load among the processors.

Phases one and four are application-dependent, phase two depends on the working platform. Step one requires to define a load measure. For MD simulations, the load can be specified as the number of molecules assigned to a processor. We have decided to report the workload at the same time that the MD algorithm recalculates the neighbors of the molecules (every twenty iterations).

At step two, the processors make the decision of whether it is convenient to balance the molecular system or not. In order to achieve the best performance, the system should tolerate imbalance levels which are related to the overhead caused by the method on the running simulation.

Step three corresponds to the load balancing operation. We work with the Generalized Dimension Exchange (G.D.E.) method [14] that can be classified as a deterministic nearest-neighbor load balancing algorithm. In this method, once the processors have decided to initiate the load balancing operation, each processor compares its workload with the load of its nearest neighbors and computes the portion of load difference to exchange. This operation must be repeated until the workload information about the processors is propagated through all the network. At this point two questions arise: how much load must each process give/receive; how many times must the exchange be done. The answers to both questions depend on the topology of the network. We have chosen a chain topology with even number of processors in which the exchange factor is 0.5 and the procedure must be repeated $\frac{\text{diameter of the chain}}{2}$ times [14]. Other topologies are possible but in general they have worse exchange factors or they have not known convergence measurements.

Step four implies to redefine the space assigned to processors. The molecules are increasingly ordered according to their distance to the actual border using a bubblesort. Ordering is done until the amount of molecules achieves the

amount of load to move. The action of moving a molecule from a processor to another involves a change in the workload of the processor. Finally, when the right amount of molecules have been found, the border of the sub-domain space is moved and the molecular neighbors are recomputed.

4 Profitability Determination of Load Balancing for MD Simulations

The choice of the right moment for load balancing is decisive to achieve an improvement of MD simulations. Load balancing must be carried out only when the performance benefit is greater than the overhead cost. Thus, tolerable load imbalances below a threshold are useful to improve performance. Here, we are interested on determining a threshold for MD simulations and we present an analytical model to determine when it is profitable to balance the system.

4.1 Tolerable and Critical Unbalance Levels

J. Watts and S. Taylor [12] state the goal of load balancing as follows:

Given a collection of tasks comprising a computation and a set of computers on which these tasks may be executed, find the mapping of tasks to computers that results in each computer having an approximately equal amount of work.

For MD simulations, a measure of the amount of work for a processor is the number of molecules it has [3]. A linear increase of molecules causes a quadratic increase of the amount of work. Thus, when the molecules migrate from one processor to another the unbalance problem might be aggravated. However, if our goal is to improve the execution time of the application, not every unbalance must be corrected. Every twenty iterations the MD algorithm completes a cycle which includes the computation of forces, the change of position and velocity, and the redistribution of the molecules among the processors. Thus, we select a time slice of twenty iterations to compare the times involved in MD simulations.

Given a system with p processors and a MD simulation of n molecules, for an execution cycle the optimal execution time (T_{opt}) is achieved when every processor has $\frac{n}{p}$ molecules. Let us suppose that n_{max} is the number of molecules of the processor $P_{n_{max}}$ in a cycle with the maximum load of the system. The execution time of the cycle is $T(P_{n_{max}})$. If n_{max} is $\frac{n}{p} + \alpha$ where α is small, then $T_{opt} \approx T(P_{n_{max}})$. In such case, we say that the system presents a *tolerable unbalance level*. The *critical level* of a system is achieved when the difference between the actual execution time ($T(P_{n_{max}})$) and the optimal time (T_{opt}) is greater than the overhead time of the load balancing algorithm ($T_{overheadLB}$). In this case, it is convenient to trigger the load balancing algorithm. This can be expressed as follows,

$$T(P_{n_{max}}) - T_{opt} > T_{overheadLB} \quad (1)$$

Table 1. Overhead of the Load Balancing Operation for Different Tolerance Levels

| Tolerance level (%) | step 1 (%) | step 2 (%) | step 3 (%) | step 4 | | |
|---------------------|------------|------------|------------|---------------|--------------------|----------------------|
| | | | | sort time (%) | migration time (%) | neighboring time (%) |
| 7.5 | 2.09 | 0 | 0 | 1.29 | 0.44 | 96.16 |
| 10 | 3.58 | 0 | 0.04 | 2.19 | 0.52 | 93.64 |
| 12.5 | 15.4 | 0 | 0 | 3.08 | 0.20 | 81.31 |

In subsection 4.2 we will show a model to compute the $T_{overheadLB}$ of a MD simulation and in subsection 4.3 we will establish how to compute $T(P_{n_{max}})$ and T_{opt} .

4.2 Overhead of the Load Balancing Method Applied to MD Simulations

In Section 3 we have enumerated the steps of the load balancing operation. In the algorithm we can distinguish between two types of operations: computation and communication. There are computations of order constant ($\mathcal{O}(c)$) at steps two, three and four (when the molecules are sorted) and computations of $\mathcal{O}(n^2)$ at step four (when the neighbors are recomputed). There are communications of order constant at steps one and three and communications of linear order depending on the number of molecules $\mathcal{O}(n)$ at step four (when the molecules are moved from one processor to another).

Table 1 shows the overhead measures obtained for a load balancing operation in a MD simulation running on four processors with different tolerance levels.

In Table 1 we observe that the overhead of the load balancing method is due mainly to the recomputation of neighbors at step four. Sorting of molecules is done using a bubblesort that stops when the amount of molecules to move (m) is already sorted ($m \ll n$). The algorithm requires $\frac{3m}{4}(2n - m - 1)$ movements in average. The migration time depends on the amount of molecules to exchange among the processors, the network latency and bandwidth. The contribution of the sorting and migration operations to the overhead is not significant and it will be discarded in a first approach of the model.

The neighbor computation time consumes most of the overhead time. Once the balance is achieved, the neighbor of the molecules is recomputed. That is, $T_{neigh} = T_{neigh}(P_{n/p})$. $T_{neigh}(P_{n/p})$ is a measure obtained at the beginning of the execution of the application.

In conclusion, the overhead of the load balancing for MD simulations can be computed as:

$$T_{overheadLB} = \epsilon + T_{neigh}(P_{n/p}) \quad (2)$$

Table 2. Comparison of Execution Times

| | No Load Balance | 12.5% Unbalance | Profitability Determination Using the Model |
|-----------------------------|--------------------|--------------------|--|
| Total Execution Time (secs) | 44987 | 42370 | 39775 |
| Time Improvement (%) | - | 5.82 | 11.59 |

4.3 Overhead of a MD Simulation Due to the Unbalance

The extra computation time in a MD simulation is the difference between the actual computational time in an execution cycle (equal to the execution time of the more loaded process) and the computational time of a cycle for a system well balanced. That is, $T_{overheadMD} = T(P_{n_{max}}) - T_{opt}$. The $T(P_{n_{max}})$ is a measure provided by the heaviest loaded processor and T_{opt} is a measure obtained at the beginning of the execution of the application.

From Equations 1 and 2 we have that the critical unbalance condition is achieved when Equation 3 is satisfied.

$$T(P_{n_{max}}) - T(P_{n/p}) > T_{neigh}(P_{n/p}) \quad (3)$$

5 Simulation Experiments

We tested the G.D.E. algorithm customized to the M.D. problem. The benchmark has been adapted from the one presented by S. Plimpton [10]. It consists of 32000 molecules simulated in a 3D parallelepiped with periodic boundary conditions at the Lennard-Jones state point defined by the reduced density $\rho^* = 0.3$ and reduced temperature $T^* = 0.8$. Since this is a homogeneous substance, it does not exhibit significant unbalance levels. In order to perform our experiments we forced the unbalance by changing the substance temperature. At the iteration 70000, the substance was cooled until $T^* = 0.005$, and in the iteration 140000 it was heated until $T^* = 0.5$. The time measurements were taken between the iterations 120000 and 270000.

The experiments were carried out using 4 nodes of a Pentium III cluster with double processor nodes running at 667 MHz. Each node has 512MB of RAM and a separate instruction and data cache of 16KB. The nodes are connected to each other through Myrinet. The message-passing environment used is MPI.

We compare the performance of the G.D.E. method on two different situations. The first one presents the behavior of the load balance method using an ad hoc trigger condition and the second one using the analytical model presented in Section 4. Table 2 shows that the best time improvement is achieved when the balance decision is made following the analytical model proposed. The trigger condition chosen by the model occurred when the unbalance of the system was 17.8%.

6 Conclusions

In this paper, we have shown that a distributed load balance method such as G.D.E. is suitable for MD systems since it improves the performance of MD simulations.

An important issue of any dynamic load balance algorithm is to decide the appropriate moment to balance. If the system is balanced too often then the improvement achieved could be lost by the load balancer overhead. Therefore, the selection of an initiation rule is crucial in load balance. This selection could be a static or dynamic decision. In this paper, we have compared the performance of a G.D.E. load balancer using an ad hoc decision and a dynamic trigger condition based on the analytical model proposed at Section 4. Our experiments have shown that the application of the analytical model improves significantly the performance of the simulations.

In this first approach we applied a simplified version of our analytical model. Currently, we are concentrating our efforts on including the sort and migration time of the load balancer that we discarded in this simplified version.

The analytical model can be easily extended to a heterogeneous platform such as the ones found on cluster architectures.

References

- [1] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive Load Balancing Policies for Dynamic Applications. *IEEE Concurrency*, pages 22–31, January–March 1999. 227
- [2] A. Cortes, A. Ripoll, M. A. Senar, P. Pons, and E. Luque. On the Performance of Nearest Neighbors Load Balancing Algorithms in Parallel Systems. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing*, Funchal, Portugal, February 1999. 227
- [3] A. Di Serio and M. B. Ibáñez. Distributed load balancing for molecular dynamic simulations. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications*, Moncton, New-Brunswick, Canada, June 2002. IEEE Computer Society. 229
- [4] K. Esselink, B. Smit, and Hilbers. Efficient parallel implementation of molecular dynamics on a toroidal network: Multi-particle potentials. *Journal of Computer Physics*, 106:108–114, 1993. 227
- [5] Finchman. Parallel computers and molecular simulation. *Molecular Simulation*, 1(1-45), 1987. 226
- [6] J. M. Haile. *Molecular Dynamics Simulation*. Wiley Inter-Science, 1992. 226
- [7] D. F. Hegarty and T. Kechadi. Topology Preserving Dynamic Load Balancing for Parallel Molecular Simulations. In *Super Computing*, 1997. 227
- [8] L. V. Kalé, M. Bhandarkar, and R. Brunner. Load Balancing in Parallel Molecular Dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, 1998. 227
- [9] M. R. S. Pinches, D. J. Tildesley, and W. Smith. Large-scale molecular dynamics on parallel computers using the link-cell algorithm. *Molecular Simulation*, 6(51), 1991. 226

- [10] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Of Computational Physics*, 117:1–19, 1995. 226, 227, 231
- [11] N. Sato and J. M. Jézéquel. Implementing and Evaluating an Efficient Dynamic Load-Balancer for Distributed Molecular Dynamics Simulation. In *Proceedings of the 2000 International Workshops on Parallel Processing*. IEEE, 2000. 227
- [12] Jerrell Watts and Stephen Taylor. A Practical Approach to Dynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235–248, March 1998. 227, 229
- [13] M. Willebeek-LeMair and A. Reeves. Strategies for Dynamic Load Balancing on High Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4:979–993, 1993. 227
- [14] C. Xu and F. C. M. Lau. *Load balancing in parallel computers. Theory and Practice*. Kluwer Academic Publishers, 1997. 227, 228

Application Recovery in Parallel Programming Environment^{*}

Giang Thu Nguyen¹, Viet Dinh Tran¹, and Margareta Kotocova²

¹ Institute of Informatics, SAS
Dubravska cesta 9, 84237 Bratislava, Slovakia
`giang.ui@savba.sk`

² Department of Computer Science, FEI STU
Ilkovicova 3, 81219 Bratislava, Slovakia

Abstract. In this paper, fault-tolerant feature of TOPAS parallel programming environment for distributed systems is presented. TOPAS automatically analyzes data dependence among tasks and synchronizes data, which reduces the time needed for parallel program developments. TOPAS also provides supports for scheduling, load balancing and fault tolerance. The main topics of this paper is to present the solution for transparent recovery of asynchronous distributed computation on clusters of workstations without hardware spare when a fault occurs on a node. Experiments show simplicity and efficiency of parallel programming in TOPAS environment with fault-tolerant integration, which provides graceful performance degradation and quick reconfiguration time for application recovery.

1 Introduction

Advances in information technologies have led to increased interest and use of clusters of workstations for computation-intensive applications. The main advantages of cluster systems are scalability and good price/performance ratio, but one of the largest problems in cluster computing is software [5]. PVM [14] and MPI [15] are standard libraries used for parallel programming for clusters and although they allow programmers to write portable high-performance applications, parallel programming is still difficult. Problem decomposition, data dependence analysis, communication, synchronization, race condition, deadlock, fault tolerance and many other problems make parallel programming much harder.

TOPAS (Task-Oriented PARallel programming System, formerly knows as Data Driven Graph - DDG [9][10][11]) is a new parallel programming environment for solving the problem. The objectives of TOPAS are: making parallel programming in TOPAS as easy as by parallel compilers, with the performance comparable with parallel programs written in PVM/MPI; making parallel programs structured, easy to understand and debug, and to allow error checking at

^{*} This work is supported by the Slovak Scientific Grant Agency within Research Project No. 2/7186/20

compilation time for removing frequent errors; providing support for optimization techniques (scheduling and load balancing); providing facilities for Grid computing (heterogeneous architectures, task migration, fault tolerance). Environment description, scheduling and load balancing in TOPAS are already presented in [9][10][11]. In this paper, we propose a method to provide a fault-tolerant feature combining with task reallocations for parallel and distributed programs. The checkpointing and recovery process is made in such way that the application achieves as small overhead as possible. The target platform of the method is represented by distributed-memory systems with message-passing communication such as cluster systems.

The rest of the paper is organized as follows: section 2 shows the TOPAS main features. Section 3 focuses on fault tolerance support and section 4 demonstrates real examples of parallel programs written in fault-tolerant TOPAS. A conclusion is in Section 5.

2 TOPAS Environment Main Features

Parallel programming is developed through six steps: problem decomposition, data dependence analysis, task graph generation, scheduling, data synchronization and coding tasks. Except for coding tasks (similar to sequential programming), the other five steps can be divided into two groups. The first group (decomposition and scheduling) is important for performance and every change in these steps may affect performance largely. There are many possible solutions and no general way to choose the best one. Experienced programmers often do these steps better than parallel compilers. Each of other three steps (data dependence analysis, data synchronization and generating task graphs) has only one solution and simple algorithms exist to get it. These steps have little effect on performance (note that inter-processor communication is minimized by problem decomposition and scheduling) and require a lot of work. Most frequent errors in parallel programming occur here.

In TOPAS, the performance critical steps (problem decomposition and problem scheduling) are done by programmers and the remaining is done automatically by TOPAS runtime library. This way, writing parallel programs written in TOPAS is as easy as by parallel compiler, while the parallel programs in TOPAS can achieve performance comparable with parallel programs in PVM/MPI.

The architecture of TOPAS library is shown in Fig. 1. The core of TOPAS is the TOPAS kernel that manages all tasks and data. Communication module creates interface between the kernel and underlying communication libraries (PVM, MPI, Nexus) and make parallel programs in TOPAS independent from communication library. TOPAS API is the application-programming interface, which allows programmers to create tasks and variables. Other modules (scheduling, reconfiguration) use the memory structures in the kernel.

The basic units of parallel programs in TOPAS are tasks. Each task is a sequential program segment that can be assigned to a processor node; once it starts, it can finish without interaction with other tasks. To extend, tasks

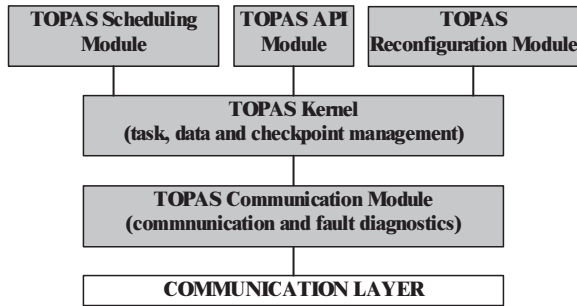


Fig. 1. TOPAS architecture

can be understood as the sequential parts between two communication routines in PVM/MPI. Each task in TOPAS consists of a function/subprogram that contains task code and a set of variables the task uses. Tasks are created by calling the *create_task()* function with its code and its variables as parameters. Each variable may have read-only (*ro*), read-write (*wr*) or write-only (*wo*) access. Each task is represented by a task object that has pointers to its code and its variables. Using the memory structure, TOPAS runtime library can detect the data dependence among tasks and synchronize data automatically. If several tasks write to the same variable, then multi-version technique [13] is used to solve the data dependence among tasks. Each task that writes to the variable creates a new version of the variable. Versions are created dynamically when needed, and they are removed from memory if there are no unexecuted tasks that use the version [10]. In any case, the complexity of data dependence analysis and creating internal memory structures of programs in TOPAS is proved to be linearly increasing with the number of tasks.

Scheduling is one of the most important techniques to improve the performance of parallel programs [2][3][4][6][8]. When a parallel program in TOPAS runs, after tasks have been created, TOPAS kernel calls *ddg_schedule()*, which receives the task graph as input parameter and returns a generated schedule as output parameter. Default scheduling in TOPAS library is done by Dominant Sequence Clustering [7] algorithm.

3 Integrating Fault-Tolerant Feature into TOPAS

As the size of computer systems increases unsteadily, the probability that faults may occur in some nodes of the system increase as well. Therefore, it may be necessary to ensure that the applications may continue in the system despite the occurrence of faults [12]. This is very important in Grid computing [1], as computational nodes are not managed by one owner and communications via the Internet are unreliable. Traditional system recovery applies in fixed order: fault detection and location, FT reconfiguration, restarting system with valid data

(from the last valid checkpoints). When hardware redundancy is not available, software redundancy is unavoidable.

Data Checkpoints

The basic concept of fault tolerance in TOPAS is to keep spare instances of task output data on other nodes as checkpoints. Assume that only one single fault may occur on a node:

- One spare instance of data is sufficient, if it is required only by other tasks that are allocated on the same node. The spare instance must be created and located on a different node of the node, where these tasks are allocated. When a fault occurs on a node, one of its instances (original or spare) is still available in a healthy node.
- No spare instance is necessary, if data is required by other tasks allocated on (at least) two different nodes. In this case, instances of the output data are already sent to nodes, where these required tasks are allocated. After fault occurrence, one node will be faulty but (at least) one instance is still available in (an)other healthy node(s).

The set of spare instances is minimal. A spare data instance is created if it is unavoidable in the means of fault tolerance. Each node needs certain information about system and tasks. The information are created at runtime, during scheduling and is updated during application runtime. They are: TOPAS application data flow graph - created when the application starts; states of all tasks; and locations of checkpoints. There is a requirement to keep low number of inter-node communications, otherwise the system performance decreases significantly. If a checkpoint is created, its location must be announced to all nodes and this can cause a large number of broadcasts. Therefore, in TOPAS, spare instances are located in the neighbor node. If the output data is required by at least two tasks in different nodes, locations of all data instances are in data-flow graph of the application.

The idea is easily extended for multiple simultaneous faults at one time. Let number of possible simultaneous fault be f . For a data, it may have already n instances because tasks on n different nodes use it.

- If ($f < n$), no spare instance is needed and no additional communication is required.
- If ($f > n$), ($f + 1 - n$) additional spare instances so the total number of instances of the data is ($f + 1$); and if f faults occur simultaneously on f nodes, there will be (at least) one available (valid) instance remained. Spare instances are created in a safe place (node), which leads to centralized coordination but reduces number of instances in the system.

In TOPAS, when fault-tolerant feature is not supported, removing unnecessary output data is simple. When the fault-tolerant feature is supported, the

use of multi-version variables is unavoidable and data checkpoints are kept until they are unnecessary for all tasks on all nodes. Broadcast messages and task synchronization are frequently mentioned, but they cause a large number of inter-processor communications and/or keep large amount of obsolete data. Because of that, the fault tolerant TOPAS environment uses its own removal method. Each node has own information about executed and waiting tasks in the system. The information is collected using TOPAS data flow graph, which is available on each node. Accordingly, if a task is executed, all its predecessors are finished too; when a node receives an instance, it knows which task creates the instance. This task is executed (determined by the node) and its predecessors are executed too.

Application Reconfiguration

Reconfiguration process (Fig. 2.) is done as follows:

- New configuration is computed in the master node of reconfiguration (centralized control of reconfiguration). The master node is a floating one.
- The new configuration is sent and actualized in every healthy node (decentralized actualization of reallocated tasks and data recovery) .

After fault detection and location, each node knows the master node of reconfiguration according to node ID numbers. The master node initializes reconfiguration (allocates memory and creates the state table of tasks on all nodes). After that, it waits for incoming task states from remaining healthy nodes. Other

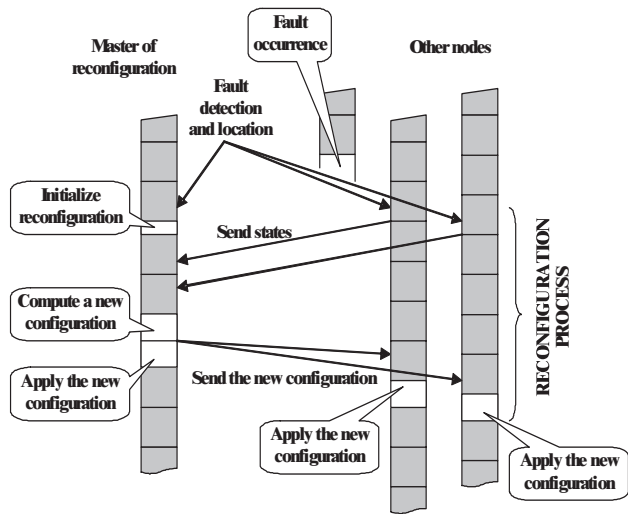


Fig. 2. Hybrid reconfiguration

healthy nodes determine that they are not the master node of reconfiguration. They also know the master node and send their current task states to it. During waiting time (for task states or new configuration), healthy nodes continue their work (e.g. receiving data, execute tasks) according to the original configuration.

After receiving task states from other healthy nodes, the master node fulfils the state table and determines states of all tasks on the system. If a task is considered as executed on one node, it is considered as executed in the whole system. Based on the original configuration and current state information, the master node computes a new task configuration (using reconfiguration module) and sends it to other healthy nodes. In general, the problem of finding optimal configuration for tasks reconfiguration is hard NP-problem. Application performance is affected by many factors: system architecture, input/output data, original scheduling, designed algorithms, etc.; then experienced users (programmers, developers) may want to provide own algorithms to achieve their best desired performance. In this case, TOPAS provides a space for user-defined reconfiguration algorithms. Otherwise, the default algorithm is used. The chosen criterion is a minimal amount of moving data due to task reallocation, so tasks on the faulty node are reallocated to its neighbor.

After obtaining the new configuration, each node realizes by itself. Each node reconciles states of tasks assigned to it and actualizes reallocated tasks. The node checks their existing input data, activates task codes, and inserts them to the waiting queue for node or for input data. After that, the node continues its work according to the new configuration.

4 Case Study

The Gaussian Elimination Method (GEM) is used as an example running in TOPAS fault-tolerant environment. All experiments are done in PC cluster of Pentiums III connected through 100Mbps Ethernet.

```

1.#define N 2400                                //Sequential version of GEM
2.main()
3.{ double a[N][N];
4.  init(a);                                     // init value of a
5.  for (int i=0; i<N-1; i++)
6.      for (int j=i+1; j<N; i++)
7.          { coef = a[j][i]/a[i][i];
8.              for (int k=i+1; k<N; k++)
9.                  a[j][k] = a[j][k]-coef*a[i][k];
10.      }
11. print(a);                                    //print result of a
12.}
```

The sequential Gaussian elimination algorithm - GEM is described above. The tasks are defined from the lines inside two outer loops (line 7, 8, 9). Before defining a task, the code of the tasks has to be moved to a function. Finally,

the task is created from the code. It can be seen in the final version of GEM in TOPAS, that there are no communication routines inside of functions as follows:

```
1.#include "ddg.h"                                //Parallel version of GEM
2.#define N 1200                                  #define GRAIN 20
3.typedef float vector[GRAIN][N];
4.void ddg_main()
5.{  ddg_var_array<vector> arr(N/GRAIN);
6.   init(a);                                     //initial the values of a
7.   for (int i = 0; i < N/GRAIN - 1; i++)         //1st for
8.       for (int j = i; j < N/GRAIN; j++)         //2nd for
9.           ddg_create_task(compute,              //3rd for
10              ddg_direct(i), ddg_ro(arr[i]), ddg_rw(arr[j]));
11.   print(a);                                    //print result values of a
12.}
```

GEM experiments in Table 1. are done for matrix size 3600x3600. The values show GEM speedup comparison in non-FT environment and in TOPAS without fault occurrence.

The speedup of Gaussian elimination algorithm in FT TOPAS is shown in Table 2. The additional overhead for fault-free execution is done as the difference between execution time in non-FT environment and execution time in FT TOPAS environment. The value in the range from 0, 2% to 4% of execution time.

GEM experiments in Table 2 are done for matrix size 2400x2400. The whole application contains of 29160 tasks, each node originally has around 7300 tasks and the application originally runs on four nodes. After the application recovery, it continues on three nodes. Note that the execution time in three nodes for such matrices without fault occurrence is around 45000 milliseconds. The execution time in four nodes without fault occurrence is around 35500 milliseconds.

Table 1. GEM speedups in non-FT and TOPAS environment

| Number of nodes | 1 | 2 | 3 | 4 | 5 | 6 |
|--------------------|------|------|------|------|------|------|
| Non-FT environment | 1,00 | 1,83 | 2,64 | 3,38 | 4,00 | 4,58 |
| TOPAS environment | 1,00 | 1,76 | 2,56 | 3,24 | 3,85 | 4,02 |

Table 2. GEM application recovery after fault occurrence. Execution time in ms

| Fault occurrence after | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 | 7000 |
|----------------------------|-------|-------|-------|-------|-------|-------|-------|
| Equally Task Reallocation | 45033 | 44211 | 43205 | 42575 | 41804 | 41504 | 39380 |
| Neighbor Task Reallocation | 60000 | 55800 | 52000 | 48000 | 44800 | 41337 | 38500 |

5 Conclusion

In this paper, TOPAS parallel programming environment is described. TOPAS does not only allow users to write parallel programs easily but also provides facilities for scheduling, load balancing and fault-tolerant features. The environment is carefully designed in effective way that provides following significant achievements from the fault-tolerant viewpoint:

- Providing the efficient method for creating checkpoints and removing obsolete data that avoids exhausting resources, additional inter processors communication, task synchronizations and checkpoint coordination.
- Minimizing the inter processor communication required by synchronizing states of tasks on all nodes. The data recovery and actualization of task reallocation is distributed so it preserves independence and parallelism in the system.
- Requiring small additional overhead. Application recovery is transparent with minimal amount of wasted computation. There is no domino effect.
- Maximizing processor utilization despite of fault occurrence.
- Fault tolerance is integrated to the environment automatically and it does not require any modifications in source code or any extra work from users. TOPAS environment provides a space for user defined algorithms to achieve their desired performance.

Experiments have demonstrated the usability of TOPAS on real problems, the simplicity of parallel programming in TOPAS, the performance of TOPAS programs and the overhead of TOPAS runtime library

References

- [1] Kennedy: The Grid - Blue Print for a New Computing Infrastructure, pp. 181-204, Morgan Kaufmann, 1999.
- [2] L. Hluchy, M. Dobrucky, D. Dobrovodsky: Distributed Static and Dynamic Load Balancing Tools under PVM. First Austrian-Hungarian Workshop on Distributed and Parallel Systems, Hungary, 1996, pp.215-216.
- [3] L. Hluchy, M. Dobrucky, J. Astalos: Hybrid Approach to Task Allocation in Distributed Systems. Computers and Artificial Intelligence, Vol.17, No.5, pp. 469-480, 1998.
- [4] Senar M. A., Cortes A., Ripoll A., Hluchy L., Astalos J.: Dynamic Load Balancing. Parallel Program Development for Cluster Computing. Nova Science Publishers, USA, 2001.
- [5] H. El-Rewini, T. G. Lewis: Distributed and Parallel Computing. Manning Publication, USA, 1998.
- [6] H. El-Rewini, H. H. Ali, T. Lewis: Task Scheduling in Multiprocessing Systems. Manning Publication, USA, 1999.
- [7] T. Yang, A. Gerasoulis: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 9, pp. 951-967, 1994.

- [8] B. A. Shirazi, A. R. Hurson, K. M. Kavi: Scheduling and Load Balancing on Parallel and Distributed Systems. IEEE Computer Society Press, 1995.
- [9] V.D. Tran, G.T. Nguyen, L. Hluchy: Data Driven Graph: A Parallel Program Model for Scheduling. Languages and Compilers for Parallel Computing LCPC'1999, pp. 494-497, USA. Lecture Notes in Computer Science, Springer-Verlag.
- [10] V.D. Tran, L. Hluchy, G.T. Nguyen: Parallel Program Model for Distributed Systems. EuroPVM/MPI'2000, pp. 250-257, Hungary. Lecture Notes in Computer Science, Springer Verlag.
- [11] Nguyen G. T., Tran V. D., Kotocova M.: TOPAS: Parallel Programming Environment for Distributed Computing. ICCS, 2002, pp. 890-899, The Netherlands. Lecture Notes in Computer Science, Springer Verlag.
- [12] M. Richmond, M. Hitchens: A New Process Migration Algorithm, Operating System Review, 31(1), 1997, pp. 31-42.
- [13] M. Bielikova, P. Navrat: An Approach to Automated Building of Software System Configurations, International Journal of Software Engineering and Knowledge Engineering, Vol. 9, No. 1, pp. 73-95, 1999.
- [14] PVM: Parallel Virtual Machine <http://www.epm.ornl.gov/pvm/pvm-home.html>.
- [15] MPI - Message Passing Interface <http://www.erc.msstate.edu/mpi/>.

IP-OORT: A Parallel Remeshing Toolkit

Éric Malouin¹, Julien Dompierre¹, François Guibault^{1,2}, and Robert Roy^{1,2}

¹ Centre de recherche en calcul appliqué (CERCA)
5160, boul. Décarie, bureau 400, Montréal, QC, H3X 2H9, Canada
{malouin,julien,francois,robroy}@cerca.umontreal.ca

² Département de génie informatique, École Polytechnique de Montréal
Case postale 6079, succ. Centre-ville, Montréal, Québec, H3C 3A7, Canada

Abstract. This paper presents the current strategy used in an ongoing project to extend the application domain of a C++ toolkit library for iterative mesh adaptation. **OORT** is a class library for sequential structured, unstructured and hybrid mesh adaptation used mainly in the context of CFD computations, that performs iterative mesh refinement, coarsening and smoothing in 3D. Extensions to parallelize mesh adaptation using ParMETIS for domain decomposition and MPI high-level communication schemes are here investigated. Numerical simulations on realistic cases show that the parallel strategy scales with problem size and the number of processors, but singular behaviors are sometimes encountered at subdomain interfaces when conflicting instructions collide.

1 Introduction

A major goal pursued in an ongoing research project carried out jointly by CERCA and a number of industrial partners including GE Hydro and Bombardier Aerospace is to develop automatic meshing tools to be used in the context of large-scale flow simulations around hydraulic and airplane components. It is hoped that these adaptive meshing tools will enable easier and more reliable comparison between solutions, taking into account the levels of computing power currently available for everyday runs.

While this project has already yielded some interesting results for many types of industrial geometries in sequential mode [1, 2], computational speed of the adaptation process has become an important issue not only in the context of this project, but more generally for many researchers involved in computational fluid dynamics (CFD). For example, see the works of T. Coupez on parallel remeshing.¹ Indeed, mesh adaptation is one among a number of important steps that make up a complete flow simulation, and while significant efforts have been made to speed up the equation solving process itself, only limited efforts have been devoted to the speeding up of adaptation processes. In most instances, parallelization of the adaptive process is presented as part of an effort to parallelize the flow solver [3, 4]. However, mesh adaptation is a complex process

¹ <http://www-cemef.cma.fr>

which involves its own set of constraints on mesh quality which are not related to a specific solving scheme but rather to the mesh topology and the set of modification operations. This paper will emphasize constraints related to structured grid topology and vertex displacement operations in the context of a tensorial size specification map stemming from error analysis.

The paper is organized as follows. Section 2 describes the global solution process including mesh adaptation, with particular emphasis given to the mesh adaptation itself. Section 3 focuses on the parallel approach taken and discusses domain partitioning and communications. Section 4 presents early parallel results obtained for structured mesh adaptation using the current strategy; numerical simulations show that distortions can occur at interfaces due to conflicting instructions from neighboring nodes. The paper closes with remarks on variations in the parallel approach that will be implemented and validated in the near future.

2 Description of the Application

This section presents a brief overview of the **OORT**² (Object Oriented Remeshing Toolkit) and **IP-OORT** (Parallel-**OORT**) libraries, their purpose and functionality.

Figure 1 illustrates a typical solution process including mesh adaptivity. This process includes the following important steps: 1) geometric modeling of the geometric model, 2) initial mesh generation from CAD data, 3) boundary conditions imposition, 4) flow solution, 5) error estimation based on the solution, that allows to determine regions where the mesh should be clustered (or loosened) because of high (or low) error estimates; error estimation results are transferred to the adaptation phase as an element size specification map, 6) mesh convergence test based on both the error equidistribution and performance of the component under analysis, 7) mesh adaptation, which actually modifies the mesh according to clustering characteristics extracted from the error estimation, and 8) solution interpolation that computes a restart solution to improve the stability and convergence of the flow solver.

This procedure is valid for any type of mesh, whether it is structured, unstructured or hybrid. Also, regardless of the type of mesh involved, important computer time expense is incurred mainly in steps 4 and 7. Step 4 deals with flow equation resolution; in the current process, this is left to the CFX-Tascflow³ commercial package. Step 7 is the mesh adaptation procedure itself, which will be considered here, mainly in the context of structured meshes.

When a 3D structured mesh is adapted, the remeshing algorithm needs to iterate over all vertices of the mesh and displace them one by one to guarantee the conformity of the elements [5]. This process is carried out over and over again, until the mesh satisfies, to a given tolerance, the size specification map

² www.cerca.umontreal.ca/oort

³ www.software.aeat.com

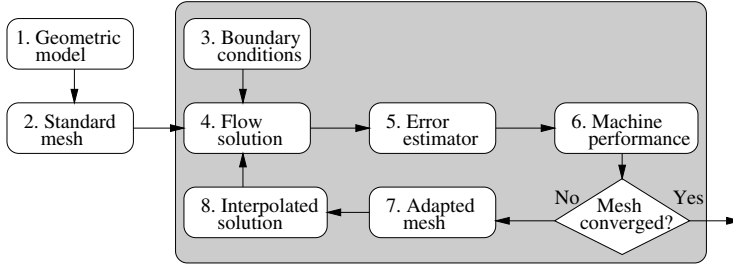


Fig. 1. Flow solution process with mesh adaptation

computed in the error estimation step. This iterative process typically takes from 20 to 500 iterations depending on mesh complexity.

In order to adapt the current mesh in the target size specification map, IP-**OORT** stores data about every vertex, edge and element and makes sure that each vertex can directly determine which edges and elements it is connected to. In the similar way, elements and edges store which vertices they are connected to. Connections between an element and its edges are not stored since this information can be recomputed simply. In the case of an unstructured mesh, this data structure is dynamically updated, as the mesh is gradually adapted towards the required size map. IP-**OORT** uses a static mesh and solution to represent the size specification map as a piecewise trilinearly interpolated tensor field.

Figure 2 presents a class diagram to describe the application. Remeshing is still in development and the methods and algorithms in **OORT** are constantly being perfected. For portability and maintenance, it is crucial that improvements made on **OORT**, the sequential version of IP-**OORT**, can also be migrated to IP-**OORT**. For this reason, IP-**OORT** needs to easily integrate the improvements found in **OORT**. Although **OORT** was written in C++, it is clear that C++ building would allow IP-**OORT** to stay as much as possible in line with **OORT**.

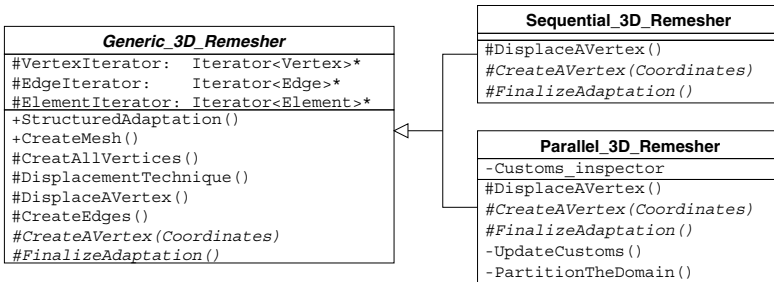


Fig. 2. UML diagram of the base class **Generic_3D_Remeshier** and the two derived classes **Sequential_3D_Remeshier** end **Parallel_3D_Remeshier**

Finally, **IP-OORT** can run on distributed memory systems such as Linux clusters, IRIX or any other POSIX-compliant architecture that can support a valid MPI implementation.

3 Parallel Strategy

This section describes the approach taken to port **OORT** into a parallel version, called **IP-OORT**. First, the high-level algorithm used by **IP-OORT** will be described with the outline of the method used to distribute the work between the different nodes. Next, how communications propagate the changes across all nodes without perturbing computations is explained, and finally ways to keep mesh adaptation specific to the local domain of every node are discussed.

The high-level algorithm used by **IP-OORT** presented in Fig. 3 for two nodes, is designed to run on a distributed memory architecture. Here are the description of the various steps necessary to achieve a global solution.

- Step A.** First the application is launched with `mpirun`. From that point, every node builds a copy of the entire mesh from the given arguments. Since nothing is created or deleted during structured adaptation, this allows **IP-OORT** to index each vertex with a unique number that it then uses as a reference across all nodes. This scheme can be assimilated to a kind of parallel iterator.
- Step B.** Once the mesh has been built on every node, step B partitions the domain into n subdomains using `ParMETIS_PartKway` [6]. From this point on, every node knows which vertices are controlled by which node.
- Step C.** **IP-OORT** then starts the inspector threads that will take care of updates that need to be done across all nodes or domains.
- Step D** **IP-OORT** starts to adapt the mesh by displacing every vertex to match as best as it can the size specification map.
- Step E.** Once it has displaced all vertices, **IP-OORT** sends a signal to its inspector so that the mesh will be updated across all nodes.
- Step F.** **IP-OORT** then starts to iterate on the whole mesh a second time.
- Step G.** The inspectors will exchange information and will update the mesh accordingly. *Note that steps E, F and G will be repeated until the required criteria is met (in step H).*
- Step H.** At this step, the convergence of the remeshing algorithm is achieved. From this point, the inspector threads will terminate.
- Step I.** Secondary nodes (from node 1 to node $n - 1$) will send their section of the domain back to the master node (node 0). The secondary nodes will then terminate while the master node will assemble the result and output it in a file.
- Step J.** Node 0 can now terminate.

In Fig. 3 (step B), the domain was partitioned using `ParMETIS`, an MPI-based parallel library that implements a variety of algorithms for partitioning and re-partitioning unstructured graphs [6]. Since a mesh can easily be described using a graph, `ParMETIS` provided all the necessary tools to partition our 3D

mesh into multiple subdomains. The function elected to partition the domain is `ParMETIS_PartKway`. It uses a multilevel k -way partitioning algorithm to partition a graph on n subdomains. Once the step B is completed, the domain is split into n subdomains with a minimal number of edges crossing more then one domain. This efficiently reduces the amount of communications required during the adaptation while keeping the load of work evenly distributed among nodes.

In Fig. 4, the process of exchanging data between nodes is sketched. As explained in Fig. 3, IP-**OORT** uses two threads during the adaptation. The main thread or *calculator* thread is used to perform all calculations while the other thread or *inspector* thread, makes sure that changes are locally done and externally updated across all nodes. This allows the mesh adaptation calculation to keep running while the communications might be blocked somewhere. In order to make sure that every vertex displaced by the calculator thread is updated, every vertex moved is stacked into a thread-safe container. Each time a signal is sent from the calculator thread to the inspector thread (see step F in Fig. 3), the inspector thread will unstack every vertex and it will decide to which node the new position needs to be sent. Once every vertex has been unstacked, the information is exchanged using `MPI_Alltoallv`, a collective communication procedure. Then the inspector will replace all ghost vertices of the ghost zone at their newly calculated position. At this point, the inspector will wait for another signal from the calculator.

Since every node has a complete copy of the whole domain, it needs to consistently know which are the vertices that are part of its own subdomain. For this reason, IP-**OORT** generates a different iterator on every node that limits the range used for adaptation. This was a key to ensure that the generic algorithm can be transparently used for adaptation in both sequential and parallel versions.

To facilitate the design of a correct implementation on several nodes with predictable performance, the bulk-synchronous parallel (BSP) model was used [7]. The parallel algorithm partitions the computation into a sequence of super-steps. In a super-step, each node independently performs its remeshing operations by iterating on a subdomain. It also collects information about load balancing distortions of the task-parallel charge occurring by destruction or creation of nodes.

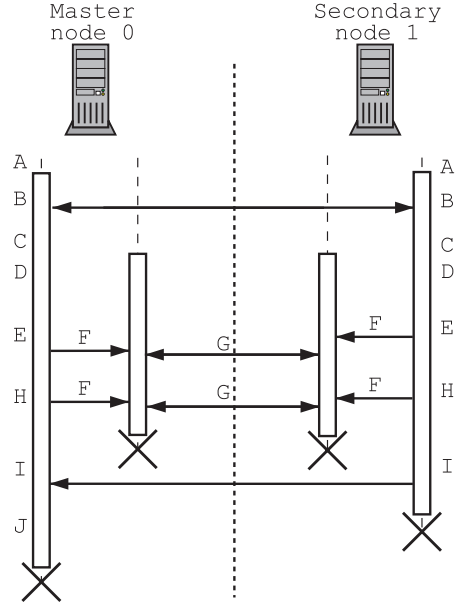


Fig. 3. UML sequence diagram of the way IP-**OORT** runs on two nodes

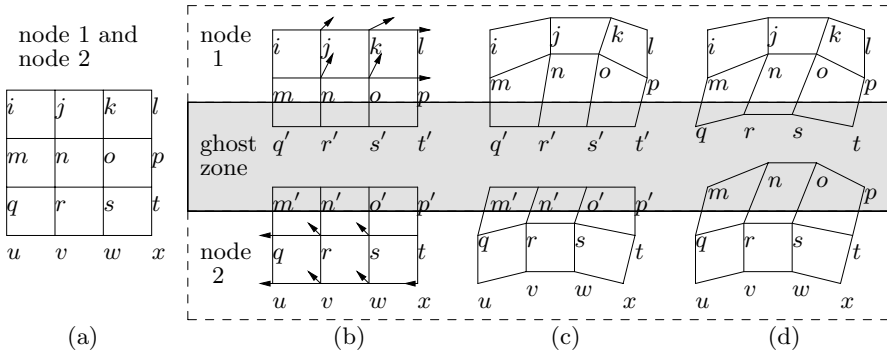


Fig. 4. Example to show how changes at the border are exchanged between nodes. Initially, node 1 and 2 have the same coordinates for all the vertices of the mesh. In step (b), each node computes the displacements that needs to be done on the vertices of its subdomain. In step (c), the result of the displacement in both subdomains are shown. In step (d), the displacement carried at step (c) are updated in the ghost zone of the direct neighbor

The message queues are built and encapsulated using MPI derived types. Synchronizations occur at the end of super-steps by calling the blocking collective communication `MPI_Alltoallv` [8]. This BSP-style of programming using the MPI standard not only simplifies a deadlock-free program structure, but also provides a simple way to adapt **IP-OORT** on different architectures by adjusting the ratio between calculation super-steps and bulk synchronizations. The generated collective communication flows are more effective than point-to-point message passing. Finally, any local memory limitations can be identified from the start of the super-step allowing for a reliable buffering protecting a node against local memory explosion.

4 Results

Tests were performed on two Linux clusters. Cluster *Xeon* has eight nodes running kernel 2.4.3. Each node has four Pentium III Xeon processors at 700 MHz sharing 4 Gb of memory. Cluster *Athlon* has 16 nodes running kernel 2.4.18. Each node has an AMD Athlon processor at 1.4 GHz with 1 Gb of memory.

The first test case has an analytic solution. The domain is a hexahedron of dimensions $[0, 1] \times [0, 1] \times [0, 2]$. The domain is split into a regular structured grid of $15 \times 15 \times 25 = 5625$ vertices and $14 \times 14 \times 24$ hexahedra (grid \mathcal{A}). This initial mesh is shown in Fig. 5(a). The solution computed on this mesh is given by the function $f(x, y, z) = \arctan(1000\sqrt[3]{xyz} - 250)$. An error estimator computed on the initial mesh is deduced from this analytical solution and the mesh is adapted with vertex relocations to fit the size specification map deduced from the error estimator as close as possible. One hundred iterations over all the vertices of

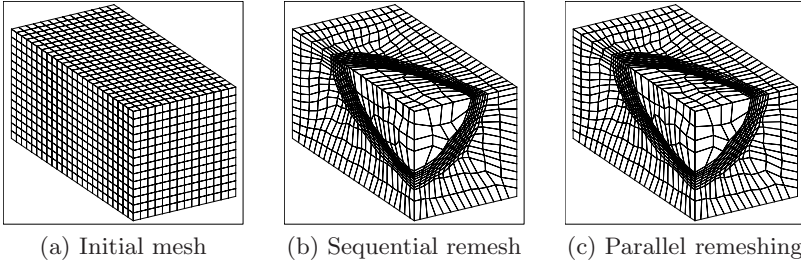


Fig. 5. Comparisons of sequential and parallel remeshing for the arctan test case

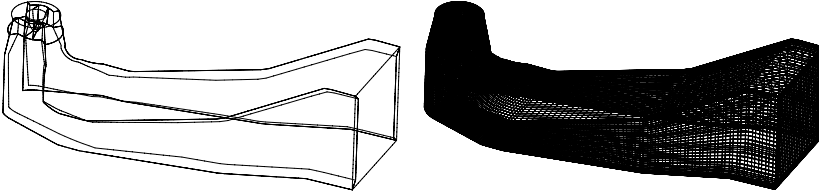


Fig. 6. Geometric model and mesh of a draft tube

the mesh were performed. The resulting mesh obtained with **OORT** is shown in Fig. 5(b) while the resulting mesh obtained with IP-**OORT** is shown in Fig. 5(c). One can see that both results are quite similar. Pictures were produced with the visualization software Vu.⁴ The same test case was done with finer and finer grids. Grid *B* has $30 \times 30 \times 50 = 45000$ vertices, grid *C* has $60 \times 60 \times 100 = 360000$ vertices and grid *D* has $84 \times 84 \times 140 = 987840$ vertices.

The second test case is a draft tube of an hydraulic turbine (Fig. 6). The grid has 7 structured blocs for a total of 127982 vertices (grid *E*). A numerical solution of the flow field was computed with CFX-Tascflow and the vertex relocation scheme is driven by an error estimator based on the interpolation error of the speed of the numerical solution.

Table 1 gives the execution time in second for **OORT** for the five test cases of this section and for the two Linux clusters. This table also gives the speed up obtained with IP-**OORT**. Results show that a Pentium III at 700 MHz is 1.75 time slower than an AMD Athlon processor at 1.4 GHz. IP-**OORT** with one processor takes 8% more time to execute than **OORT**. The more the grid has vertices (grid *A* to grid *C*), the more the speed up increases. However, with the biggest grid *D*, the speed up decreases and is even not available for 12 to 32 processors because IP-**OORT** crashed.

Even if the speed up for the grid *E* is better than the speed up of all the other grids, one can see that the speed up is roughly proportional to \sqrt{n} instead of the ideal speed up which is proportional to the number of processors n . Subdividing the domain into n subdomains should divide the execution time of the vertex

⁴ www.cerca.umontreal.ca/Vu

Table 1. Execution time in second for **OORT** and speed up for **IP-OORT**

| Grid | Cluster | OORT (s) | Number of Processors n | | | | | | | | |
|---------------|---------------|--------------------|--------------------------|------|------|------|------|----------------------------------|------|------|------|
| | | | 1 | 2 | 4 | 8 | 12 | 15 [*] /16 [°] | 20 | 24 | 32 |
| \mathcal{A} | <i>Athlon</i> | 78 | 0.92 | 1.36 | 1.92 | 2.78 | 3.25 | 3.39 [*] | — | — | — |
| | <i>Xeon</i> | 140 | 0.93 | 1.35 | 1.90 | 2.83 | 3.05 | 3.71 [°] | 3.32 | 3.29 | 3.40 |
| \mathcal{B} | <i>Athlon</i> | 483 | 0.92 | 1.30 | 2.01 | 2.67 | 3.58 | 3.46 [*] | — | — | — |
| | <i>Xeon</i> | 854 | 0.95 | 1.35 | 2.14 | 2.82 | 3.68 | 3.77 [°] | 4.64 | 4.69 | 4.84 |
| \mathcal{C} | <i>Athlon</i> | 3138 | 0.90 | 1.32 | 2.16 | 3.21 | 3.75 | 4.02 [*] | — | — | — |
| | <i>Xeon</i> | 5485 | 0.93 | 1.34 | 2.21 | 3.26 | 3.68 | 3.95 [°] | 4.19 | 4.94 | 5.25 |
| \mathcal{D} | <i>Athlon</i> | 6877 | 0.91 | NA | 2.03 | 2.74 | 3.00 | 3.52 [*] | — | — | — |
| | <i>Xeon</i> | 12224 | 0.94 | 1.20 | 2.19 | 3.02 | NA | NA [°] | NA | NA | NA |
| \mathcal{E} | <i>Athlon</i> | 868 | 0.91 | 1.51 | 2.42 | 3.54 | 4.34 | 4.83 [*] | — | — | — |
| | <i>Xeon</i> | 1475 | 0.93 | 1.56 | 2.50 | 3.80 | 4.53 | 5.52 [°] | 5.44 | 5.84 | 6.00 |

relocation scheme by a factor n . But as the the number of subdomains increases, the communication between processors needed to exchange informations about boundary vertices of the subdomains increases. The cost of the communications appears not to be negligible.

Several test cases were run with **IP-OORT**, most of which give accurate results. There are still test cases where conflicting displacements can lead to non conformal behavior. In Fig. 7, a quadrilateral in 2D and a hexahedron in 3D is non conformal if one of its corner triangle in 2D or a corner tetrahedron in 3D is flat or negative, which gives a concave element, generally rejected by finite element solvers.

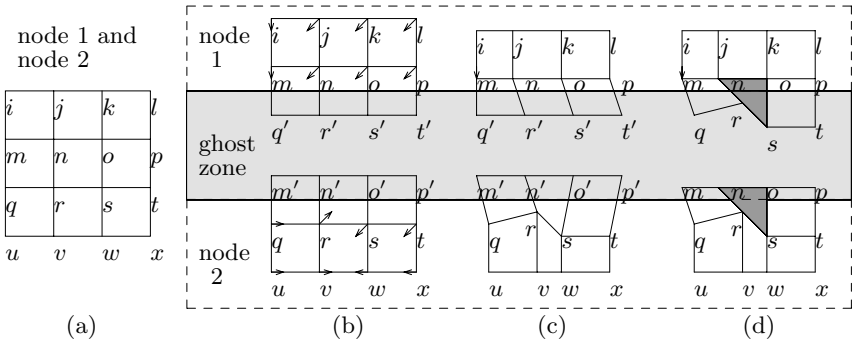


Fig. 7. Example of the apparition of a non conformal element. In step (b), each node computes the displacements that needs to be done on the vertices of its subdomain. In step (c), the result of the displacement in both subdomains are shown and quadrilateral are conformal. In step (d), the displacement carried at step (c) are updated in the ghost zone of the direct neighbor and suddenly a non conformal element (in dark grey) appears

5 Conclusion

Now that computational fluid dynamics has matured in industrial strength applications, one of the most important challenges is to develop mesh adaptation. The parallel strategy described above respects the dual opportunity to give the most portable sequential toolkit that can be optimized for day-to-day runs on PCs and to provide a scalable basis for distributing the remeshing over several nodes. Numerical simulations have shown that the wave-front of changes on each process can inconsistently collide at interfaces. Based on metrics for the particular problem, rules are locally known to tackle with displacements and other changes. The next step will be to extend these rules to subdomain interfaces, so that results are always consistent with the mesh partitioning.

Acknowledgment

We gratefully thank Sébastien Laflamme for running test cases on the various platforms. This work has been carried out partly with the help of grants from the Natural Science and Engineering Council of Canada.

References

- [1] Vu, T. C., Guibault, F., Dompierre, J., Labbé, P., Camarero, R.: Computation of fluid flow in a model draft tube using mesh adaptive techniques. In: Proceedings of the Hydraulic Machinery & Systems 20th IAHR Symposium. (2000) 243
- [2] Guibault, F., Vu, T., Camarero, R.: Automatic blocking for hybrid grid generation in hydraulics components. *International Journal on Hydropower and Dams* 5 (1999) 243
- [3] Ierotheou, C. S., Forsey, C. R., Block, U.: Parallelisation of a novel 3D hybrid structured/unstructured grid CFD production code. In Hertzberger, B., Serazzi, G., eds.: *High-Performance Computing and Networking*. Volume 919 of *Lecture Notes in Computer Science*, Springer (1995) 243
- [4] Minyard, T., Kallinderis, Y.: Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations. *Int. J. Numer. Meth. Fluids* 26 (1998) 57-78 243
- [5] Sirois, Y., Dompierre, J., Vallet, M. G., Labbé, P., Guibault, F.: Progress on vertex relocation schemes for structured grids in a metric space. In: 8th International Conference on Numerical Grid Generation, Honolulu, USA (2002) 389-398 244
- [6] Karypis, G., Schloegel, K., Kumar, V.: *PARMETIS, Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0*. University of Minnesota, Department of Computer Science and Army HPC Research Center, Minneapolis, MN. (1998) 246
- [7] Leopold, C.: *Parallel and Distributed Computing*. Wiley Inter-Science (2001) 247
- [8] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: *MPI: The Complete Reference*. MIT Press (1996) 248

Modular MPI and PVM Components

Yiannis Cotronis and Zacharias Tsiatsoulis

Department of Informatics and Telecommunications, Univ. of Athens
157 84 Athens, GR
{cotronis,zack}@di.uoa.gr

Abstract. In the Ensemble methodology message passing applications are built from separate modular components. Processes spawned from modular components specify open communication interfaces (point-point or collective), which are bound at run time according to application composition directives. We give an overview of the concepts and tools. We present and compare the design of modular PVM and MPI components.

1 Introduction

In the Ensemble methodology [1,2,3,4] message passing (MP) applications are designed and built by composing modular MP components. We have developed tools for: designing and implementing separate components; designing topologies; specifying allocation resources; generating composition directives. These tools comprise the Ensemble Composition Architecture, which has been developed on top of PVM [8] and MPI [10,11]. Ensemble designs reduce software engineering costs when compared with direct design and implementation on PVM or MPI. One advantage is that Ensemble distinguishes design and implementation issues; another is that it provides support for irregular applications, whether SPMD or MPMD. All advantages however stem from the fact that components are developed separately as independent modules and that compatible modules can be composed in any configuration without any modification.

One major task of a MP designer and programmer is to code in a program P the interactions of all processes spawned from P, in all possible positions in the process topology and for any size of the topology. Message passing APIs require specific process identifiers. The process identifiers are specified either directly (e.g. tids or ranks) or indirectly by functions, which presuppose a specific (usually regular) topology. Grids [6] impose additional requirements for program modularity, since applications may need to couple components developed by different teams. An application may be required to run independently, possibly as an SPMD (e.g. atmospheric model), or to be coupled with other applications (e.g. ocean model) running together as MPMD (e.g. climate model).

Previous Ensemble implementations [1,2,3] were aiming at developing abstract component code capable of running on any API (PVM, MPI, Parix). Consequently program components could only use a limited, semantically common, subset of

routines (e.g. send, receive, broadcast, barrier). In this paper we distinguish MPI and PVM modular components fully supporting point-point as well as collective communication in each of the two APIs.

The structure of the paper is as follows: In section 2 we present the structure of modular MPI components by example (downsized atmospheric and ocean modules); in 3 we outline PVM modular components; finally in section 4 we present our conclusions and plans for future work.

2 Modular MPI Components

Modular components are the heart of the Ensemble methodology; they are implementation abstractions of MP programs. They specify communication leaving all envelope data that is related to the origin and destination of messages (i.e. context, rank, message tag and root) unspecified. We use a data structure for storing all envelope related data, namely EnvArgs. All envelope parameters in MPI calls bind to appropriate elements of EnvArgs. For convenience we use virtual envelopes and macros, which transparently bind envelope data to EnvArgs elements. Pure MPI code is generated (envelope parameters are given proper bindings) by expanding the macros and compiled to executable modular components.

The setting of actual envelope data is done dynamically for each process. When processes are spawned their EnvArgs structure is empty. After MPI_Init each process must set values to its EnvArgs by calling SetEnvArgs, which parses and interprets (a list of) command line arguments (CLAs). The CLAs are composition directives related to a specific process and are usually generated from a tool for designing and building applications (experienced Ensemble programmers produce them directly).

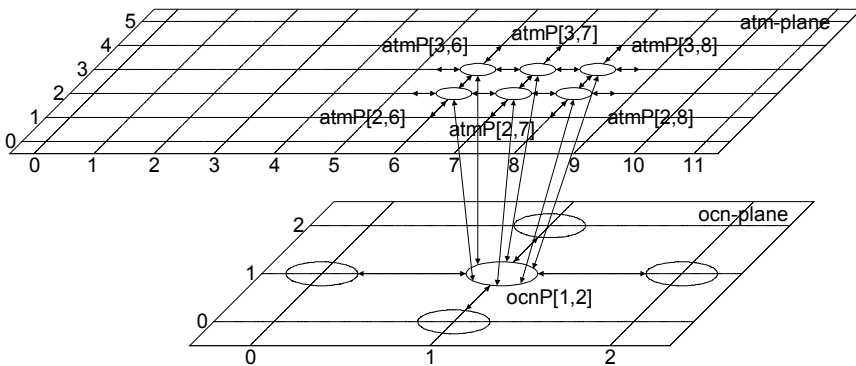


Fig. 1. The coupled climate model

In the sequel we present modular MPI components without elaborating on implementation details. We design two modular components Atm and Ocn each solving a finite difference problem, assumed to be “downscaled” versions of Atmosphere and the Ocean models respectively. Each component is required to run separately as SPMD, but also may be coupled together for solving the climate model.

Our aim is to design the Atm and Ocn components so that, on the one hand they may execute independently and, on the other to be coupled in any desirable configuration, either because of land and ocean constraints or load balancing considerations. Although, we can assume a rectangular shape for the Atm processes, this may not be true for Ocn processes, as it depends on the actual ocean-land boundaries. For this purpose we may associate Ocn processes with a variant number of Atm processes.

In figure 1 we show a possible configuration of coupling Atm and Ocn processes together solving the climate model for a given region. In the two planes we depict the two distinct SPMD applications. Only Atm processes in the eastern part of atm-plane are coupled with Ocn processes, assuming that the western region does not correspond to ocean but land. We also note that in the region where Atm and Ocn processes are coupled there is a one-six correspondence. One Ocn corresponds to six Atm processes; this is a design decision aiming to balance the load, as the atmosphere model is computationally more demanding than the ocean model.

2.1 Ensemble Atm and Ocn Components

The virtual envelope of Atm requires two contexts, Atm for processes involved in the atmospheric calculations and context X for the co-ordination of atmospheric process with corresponding ones, e.g. ocean or land calculations. Context X may involve ocean or land or be NULL; its value will depend on the application configuration. Respectively the Ocn component requires two contexts, Ocn for the ocean model calculation and Y for possible coupling with corresponding Atm processes (or some other for testing).

The actual context of each process spawned from a component will be determined at run time. As communicator types cannot be passed directly as CLAs, we use integers, which SetEnvArgs interprets as color values and creates corresponding communicators by splitting MPI_WORLD_COMM. Processes given the same color will be in the same context. The same color may be passed to processes spawned either from the same component (SPMD) or from different components (MPMD). In the former case the virtual context of the single component involved will refer to the same actual context, but in the latter the virtual context of each of the components involved will refer to the same actual context. Also processes spawned from the same component may be organized into different contexts. In this case the virtual context will not correspond to a single actual contexts; from each process's point of view however its virtual context will correspond to exactly one actual context. The composition directives determine actual contexts. In the configuration of figure 1 we have two contexts atm-plane and ocn-plane corresponding to contexts atm and ocn respectively. The groups of these two contexts consist of atm and respectively ocn processes. But groups of six atm processes with one ocn process will be in their own context, associated with some color C; the X virtual context of the six atm processes and the Y of the ocn process will refer to the actual context created from color C. If the context of an atm process is NULL it indicates that there it is not coupled with any ocn process. A process cannot (and does not need to) determine if this is because the atmospheric model runs independently or because a process of physical constraints.

Table 1. The Virtual Envelope and Ensemble Code of Atm in MPI

| Ensemble Atm Component |
|---|
| <pre> Virtual Envelope Context Atm Ports North[0..1]; South[0..1]; East[0..1]; West[0..1] Context X Ports Down [0..1] Arguments Threshold; InputFile; OutputFile </pre> |
| <pre> Code /* Declarations omitted */ MPI_Init(&argc, &argv); SetEnvArgs(&argc, &argv); while (!done) { MPI_Isend(NData,N,MPI_Float,ENVPort(North,1,Atm),&S[0]); MPI_Isend(SData,N,MPI_Float,ENVPort(South,1,Atm),&S[1]); MPI_Isend(EData,M,MPI_Float,ENVPort(East,1,Atm),&S[2]); MPI_Isend(WData,M,MPI_Float,ENVPort(West,1,Atm),&S[3]); MPI_Irecv(NData,N,MPI_Float,ENVPort(North,1,Atm),&R[0]); MPI_Irecv(SData,N,MPI_Float,ENVPort(South,1,Atm),&R[1]); MPI_Irecv(EData,M,MPI_Float,ENVPort(East,1,Atm),&R[2]); MPI_Irecv(WData,M,MPI_Float,ENVPort(West,1,Atm),&R[3]); MPI_Waitall(4,&R,&RecvStatus); AtmComputations(&LErr); MPI_Allreduce(&MaxErr,&LErr,1,MPI_Float,MPI_MAX, ENVComm(Atm)); if (MaxErr < threshold) done=true; /* Possible interactions with X (e.g.Ocean, Land) */ if (ENVComm(X) !=MPI_NULL){ MPI_Send(&Done,1,MPI_INT,ENVport(down,1,X)); MPI_Recv(&Otherdone,1,MPI_INT,ENVport(down,1,X),&st); Done = Done && OtherDone; if (!Done){ MPI_Isend(BDat,L,MPI_FLOAT,ENVport(Down,1,X),&SD); MPI_Irecv(BDat,L,MPI_FLOAT,ENVport(Down,1,X),&RD); MPI_Wait(&RD,&RDstatus); MPI_Wait(&SD,&SDstatus); } }/* End of Interactions with X */ MPI_Waitall(4,&S,&SendStatus); }/* while not done */ </pre> |

Within a virtual envelope roots for reductions and ports for point-point interaction are defined. A port is an abstraction of the envelope triplet (context, rank, message tag). Virtual ports with the same semantics are treated as an array of ports (MultiPort).

In the Atm group there are four multiports; each process may have zero or one port depending on its position on the plane. Note that in context Y of Ocn multiport Up may have up to N ports, the number of corresponding Atm processes.

Following the virtual envelope we specify arguments, which correspond to command line arguments needed in the calculations. In the case of Atm and Ocn components they specify I/O files and the criterion for termination (threshold). When the two models are coupled the same value must be passed to all atm and ocn processes, otherwise the program will deadlock.

Table 2. The Virtual Envelope and Ensemble Code of Atm in MPI

| Ensemble Ocean Component | |
|--------------------------|--|
| Virtual Envelope | Context Ocn Ports North[0..1]; South[0..1]; East[0..1]; West[0..1] Context Y Ports Up[0..N] |
| Arguments | Threshold; InputFile; OutputFile |
| Code | <pre> /* Declarations omitted */ MPI_Init(&argc, &argv); SetEnvArgs(&argc, &argv); while (!done) { /*Internal Ocean Communications similar to Atm*/ OcnComputations(&LErr); MPI_AllReduce(&MaxErr, &LErr, 1, MPI_Float, MPI_MAX, ENVComm(Ocn)); if (MaxErr < threshold) done=true; /* Possible interactions with Y (e.g.Atm) */ if (ENVComm(Y) != MPI_NULL) { for (i=1; i<ENVportN(Up, Y); i++) { MPI_Send(&Done, 1, MPI_INT, ENVport(Up, i, Y)); MPI_Recv(&OtherDone, 1, MPI_INT, ENVport(Up, i, Y), &st); Done = Done && OtherDone; } if (!Done) { for (i=1; i<ENVportN(Up, Y); i++) { MPI_Isend(T[i], L, MPI_FLOAT, ENVport(Up, i, Y), &U[i-1]); MPI_Irecv(T[i], L, MPI_FLOAT, ENVport(Up, i, Y), &Q[i-1]); } MPI_Waitall(ENVportN(Up, Y), &Q, &Qstatus); MPI_Waitall(ENVportN(Up, Y), &U, &Ustatus); } /* End of Interactions with Y */ MPI_Waitall(4, &S, &SendStatus); } } /* while not done */ </pre> |

The code of Atm and Ocn components looks like an MPI program, but all envelope-related arguments in MPI communication and synchronization calls are expressed as macros referring to virtual envelope names. Macros in atm and ocn code are shown in *italics*. All other arguments have the usual bindings. All envelop arguments of point-point communication (Rank, Message Tag, Communicator) refer to virtual envelope names by a macro e.g. ENVPort(North, 1, Atm). This macro refers to port 1 of multiport North within the Atm group. Macro ENVComm(X), refers to the actual communicator corresponding to virtual context X. A third macro ENVroot(Vcomm, Vroot), which is not used here, refers to roots. Finally ENVportN(Up, Y) corresponds to the value of ports in Up. With these macros MPI calls refer to virtual envelope names and upon expansion generate proper MPI bindings. Thus all possible communication is expressed in the code (types, order and number of messages), but no information about the receiver or the originator of

messages. The code resembles the task/channel model [5], but in a two stage manner. Stage one within component code (task to port) and stage 2 specified in the composition (port-port, context and root binding). In a way we have extended the task/channel model to deal with contexts and collective communications.

The composition of applications is specified in two levels: in a High Level Composition Tool in which symbolic names for processes, roots, groups, etc. are used. At this level the designer puts components together. In our example, the number of atm and ocn processes, as well as the shape of the two SPMD topologies are set; also the correspondence between atm and ocn processes, etc. This tool interprets the design and generates globus RSL [9] scripts, as low-level composition directives. Executing the RSL scripts the application is composed in MPICH-G [7,12]. More details may be found in [4].

3 Modular PVM Components

For PVM components the concept is the same as for MPI components: only capability of communication of processes is specified whereas all envelope data are left unspecified until process spawning. However, we need a different EnvArgs structure, reflecting the arguments of PVM calls and the relation of context, groups, roots, message tags and ports. Consequently, we need different set of macros, SetEnvArgs and argument list. These variations are reflected in the virtual envelope of PVM and MPI components. In the following section we present the virtual envelop of Atm and Ocn components in PVM, and only give some code segments demonstrating the use of the corresponding macros.

Table 3. Virtual Envelopes of Atm and Ocean Components in PVM

| Atm Virtual Envelope | Ocean Virtual Envelop |
|---|--|
| Port North[0..1]; South[0..1]; East [0..1]; West [0..1]; Down[0..1]; Bcast[0..1] Group Atm Reduction CalcMax Broadcast BcastMax Context DEFAULT | Port North[0..1];South[0..1]; East [0..1];West [0..1]; Up [0..N]; Bcast[0..1] Group Ocn Reduction CalcMaximum Broadcast BcastMaximum Context DEFAULT |

The virtual components in PVM just as their MPI counterparts consist of the envelope, the arguments and the source code. The main difference between the envelope for PVM and the envelope for MPI, results from the fact that PVM has not one integrated concept of group and context information whereas MPI has the communicator. In PVM groups may be constructed dynamically and operations inside the group do not need to have a common context. PVM contexts are not associated with a specific group, and may be applied independently to a communication operation by calling a routine to set the appropriate context. Furthermore collective communication requires message tags and groups. As a result, the virtual envelope for PVM specifies names of ports for point-point operations, multicast operations,

contexts and groups and within groups all relative information: group operations (reductions, broadcasts, barriers), roots, tags, etc. The virtual envelope of *Atm* and *Ocn* for PVM are presented in table 3.

Ports are specified independently whereas in the MPI virtual component ports are specified inside some context. Port *Bcast* is added here as broadcast messages in PVM are received by usual receive operations. Furthermore, a group *atm* (and respectively *ocn*) is specified, which inside contains a virtual name for the reduction and broadcast envelope (message tag). This group corresponds to the *atm* (and respectively *ocn*) communicator in MPI, but decoupled from a specific context. Only the *DEFAULT* context is specified, which is common for all components. No other contexts are required. Note that all virtual envelope names are local to the component.

The arguments for the PVM and MPI virtual components are identical (threshold and I/O files). The source code for the components is very close to PVM code. In Table 4 some PVM communication calls are shown.

Table 4. Some PVM calls and macros

| Abstract communication calls in <i>Atm</i> code |
|--|
| <pre>pvm_send(ENVPort(Down,1)); pvm_recv(ENVPort(East,1)); pvm_reduce(PvmMax,&LErr,1,PVM_INT,ENVRed(Atm,CalcMax));</pre> |

The macro *ENVPort(East,1)* refers to port 1 of multiport *East*. In contrast to its corresponding macro for MPI components, there is no context associated with the port. Similarly, the macro *ENVRed(Atm,CalcMax)* refers to the reduction envelope data (group name and message tag) specified by name *CalcMax* inside group *Atm*.

The High Level Composition Tool generates low-level composition directives for PVM, which are interpreted by a PVM program spawning all PVM processes with the appropriate command line arguments.

4 Conclusions

We presented MPI and PVM modular components under the Ensemble methodology. Ensemble modules (e.g. *atm* and *ocn*) are independent, but not necessarily independently developed. They are independent in the sense that their processes may be combined in many ways or not at all. However, their code must guarantee compatibility and correct performance. In general applications need to be modified if they are to be coupled together. Ensemble provides the architecture for developing modular components with the desired generality for composition. Components may then be composed without any further modifications. Other compatible components (e.g. *land*) may be also coupled with already existing ones.

Other benefits: tracing and debugging may use symbolic names of processes, roots and contexts (virtual and symbolic), e.g. “*atm*[2,6] sends to *ocn*[1,2] in context X”. No execution overhead is introduced, as envelope bindings are done at compile time. We currently develop a new composition environment, which manages components, executables, etc as grid resources and study formal support for

compatibility and composition. Our future plans are to re-engineer SPMD programs as modular components; also to address dynamically configurable applications by modifying EnvArgs during execution.

Acknowledgment

This work has been partially supported by the Special Account for Research of the University of Athens.

References

- [1] Cotronis, J.Y. (1996) Efficient Composition and Automatic Initialisation of Arbitrarily Structured PVM Programs, in Proc. of 1st IFIP International Workshop on Parallel and Distributed Software Engineering, Berlin, 74-85, Chapman & Hall.
- [2] Cotronis, J.Y. (1997) Message Passing Program Development by Ensemble, Proc. PVM/MPI'97, LNCS 1332, 242-249, Springer.
- [3] Cotronis, J.Y. (1998) Developing Message Passing Applications on MPICH under Ensemble, in Proc. of PVM/MPI'98, LNCS 1497, 145-152, Springer.
- [4] Cotronis, J.Y. (2002) Modular MPI Components and the Composition of Grid Applications, Proc. PDP 2002, IEEE Press pp 154-161.
- [5] Foster, I. (1995) Designing and Building Parallel Programs, Addison-Wesley Publishing Company, ISBN 0-201-57594-9.
- [6] Foster, I., Kesselman, C. (eds.) The Grid, Blueprint for the New Computing Infrastructure, Morgan Kaufmann, 1999.
- [7] Foster, I., Geisler, J., Gropp, W., Karonis, N., Lusk, E., Thiruvathukal, G., and Tuecke S.: Wide-Area Implementation of the Message Passing Interface, Parallel Computing, 24(12):1735-1749, 1998.
- [8] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) PVM 3 User's guide and Reference Manual, ORNL/TM--12187.
- [9] Globus Quick Start Guide, Globus Software version 1.1.3 and 1.1.4, February 2001. www.globus.org.
- [10] Gropp, W. and Lusk, E. (1999) User's Guide for mpich, a Portable Implementation of MPI, ANL/MCS-TM-ANL-96/6 Rev B
- [11] Message Passing Interface Forum (1994) MPI: A Message Passing Interface Standard.
- [12] MPICH-G2, http://www.hpclab.niu.edu/mpi/g2_body.html.

Communication Infrastructure in High-Performance Component-Based Scientific Computing^{*}

David E. Bernholdt, Wael R. Elwasif, and James A. Kohl

Computer Science and Mathematics Division, Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367, USA
{bernholdtde,elwasifwr,kohlja}@ornl.gov

Abstract. Component-based programming has been recognized as an effective technique to manage the increasing complexity of high performance scientific code. Yet the adoption of the component approach introduces new challenges that are unique to the parallel and distributed high performance scientific computing domain. The Common Component Architecture (CCA) is an ongoing effort to develop a component model targeted specifically to the needs of high-performance scientific computing, and to study the issues involved in developing, deploying, and using such an infrastructure in the HPC environment. In this paper, we present an overview of our investigation into incorporating message passing systems, namely MPI and PVM, into CCA component-based applications and frameworks. We discuss the architectural and performance issues of different design options, and their impact on developing new components as well as on the process of componentizing existing codes. These ideas are based on experiences and insights gained from developing a number of scientific simulations within the prototype parallel Ccaffeine framework.

Keywords: Scientific computing, Common Component Architecture, Component-based systems, SCMD programming, PVM, MPI.

1 Introduction

In some ways, high-performance scientific computing is a victim of its own success. The ability to simulate physical phenomena in a scientifically useful way leads to demands for more sophisticated simulations with greater fidelity and complexity. At the same time, the supercomputers on which such simulations are run grow ever more powerful, but simultaneously more complex. Combining the support of a range of such modern architectures with the increasing demands from scientific simulation experiments can lead to nearly unmanageable complexity in the software created for state-of-the-art computational science.

^{*} Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, U. S. Department of Energy under contract no. DE-AC05-00OR22725 with UT-Battelle, LLC.

The concept of component-based software development is becoming increasingly popular as a means of managing the complexity of large software systems. In the “business” and “internet” areas, tools such as the Object Management Group’s CORBA [7], Microsoft’s COM and DCOM [12], and Sun’s Enterprise JavaBeans [9] are prominent “commodity” component environments. Visualization systems such as Advanced Visualization System’s AVS, OpenDX (derived from IBM’s Data Explorer), and VTK also have a component flavor to them, with the connections between components typically representing data flow. Unfortunately, these environments do not address the requirements of high-performance scientific computing in various ways, and have seen very limited use within that community. Efforts within the community to develop component environments have mostly focused on specific problem domains, and tend to lack the generality and flexibility for a much broader user base.

In response to this situation, a grassroots effort was launched by researchers in several U.S. DOE national laboratories and partner universities to create a component environment suited to the general needs of high-performance scientific computing. The resulting “Common Component Architecture” (CCA) [1, 4] is now at the prototype stage [3, 11], and is being adopted by a wide variety of scientific computing projects.

The CCA approach to maintaining high performance is to “stay out of the way” of performance-critical operations to the greatest extent possible. Because interprocessor communication fundamentally dictates performance in most parallel computations, the CCA philosophy gives components the flexibility to use whichever communication system they prefer, including PVM [5] and MPI [13, 6]. To preserve single-processor performance, the component frameworks in CCA also support *direct* connections among components to invoke functions or methods. Local interactions among components in the same address space proceed near the speed of traditional function calls (see below).

Nevertheless, the introduction of a component environment poses some challenging obstacles for utilizing these familiar messaging tools. The authors have assisted in the development of several scientific simulation applications using the prototype parallel Ccaffeine framework [3], as demonstrated on the SC 2001 conference exhibit floor [11]. The experiences and insights gained from these preliminary exercises, in conjunction with the extensive background of the authors in the development and use of PVM and MPI, have culminated in the concepts presented in this paper.

This paper will raise and explore some of the key issues involved in integrating communication infrastructure such as PVM and MPI into component-based scientific software in the hope of establishing a dialog between developers of messaging libraries and scientific component software. These issues will be examined in the context of existing or legacy codes as well as for new scientific software development. Section 2 describes the CCA component model in more detail, followed by an examination of challenges with messaging and components in Sections 3 and 4. Section 5 will specifically discuss issues related to porting existing codes that use messaging systems into the component world.

2 An Overview of the Common Component Architecture

A number of requirements are central to the ongoing design and development of the Common Component Architecture:

- **Performance:** Its use should impose a negligible performance penalty.
- **Portability:** Support variety of scientific computing languages/platforms.
- **Flexibility:** Support broad range of parallel programming paradigms (SPMD, multi-threaded, and distributed models).
- **Integration:** Minimal requirements on existing software for use as components.

At the heart of the CCA is the concept of *ports*, through which components interact with each other and with the encapsulating framework. A port is a means of specifying and collecting together a set of publicly exported functions and methods for exchange among components. These well-defined interfaces separate semantics and invocation syntax from implementation issues. CCA ports correspond to *interfaces* in Java, or *abstract virtual classes* in C++, and follow the “uses-provides” design pattern of CORBA. There are two sides to any port connection, representing the “server” and “client” sides of port invocation. A component which implements a given interface is said to *provide* the port. Another component must declare its intent to *use* the interface *provided* by another before the framework will allow the two to be connected, and only after they are connected can the methods of the interface be invoked. It should be noted that CCA ports do *not* represent data flow, as is true in some other component systems, but rather CCA ports only indicate the capability for one component to invoke methods provided by another specific component.

The CCA *framework* is a container for holding component instances and connecting their ports together to assemble an application. Each component must declare what ports it *uses* and *provides*. This is accomplished via the component’s `setServices` method which is automatically invoked by the framework when the component is instantiated. The framework passes the component a `Services` object in the `setServices` invocation, which the component can use to register ports and obtain handles to any connected ports. The port handle is used to invoke the methods “provided” by the connected port, much as the methods in a class are invoked using a specific object instance. Because the framework provides these port handles, the CCA can provide high performance shortcuts for “local” component methods as well as remote access for distributed component methods. The `Port` object can be either an actual memory reference to a local component, or a proxy for invocation of remote component methods.

The prototype Ccaffeine framework provides an environment for *parallel* CCA component-based applications (as opposed to *distributed* applications). Components, in the form of shared objects, are loaded into distinct namespaces within a single address space (process). The use of different namespaces ensures that the components cannot interfere with each other; only the framework can “see” all components. Since the components share a single address space, the framework

can provide local components with direct references to a given port's implementation. This locally *direct* connection allows invocation of methods on another component for the cost of a C++ virtual function call – essentially a lookup of the method in the component's function table followed by the function invocation.

In parallel CCA frameworks like Ccaffeine, each instance of the framework runs in its own parallel process, and is typically loaded with an identical set of component instances and connections, in the component analog of the SPMD programming model. A collection of similar parallel component instances is referred to as a *cohort*, and the components in a cohort can communicate with each other using any available communication system, (*i.e.* MPI, PVM, Global Arrays [10], or shared memory). The CCA model of parallelism is that each framework instance mediates interactions among the components within its process (just as in the sequential case), while parallel communications within a cohort are up to the components themselves; different components can even use different systems simultaneously in a single framework. This flexibility is a major advantage to facilitating the integration of existing code into the CCA environment, however there are ramifications to the use of traditional communications systems in such a component environment (see Section 3).

To make use of components independent of their implementation language, the CCA has adopted the Babel language interoperability tool [2]. Using the Scientific Interface Definition Language (SIDL) to specify port interfaces, the Babel compiler converts the SIDL specification into glue code and header files for interlanguage invocations. Babel currently supports C, C++, Fortran 77, Python, and client-side Java, with support for server-side Java and Fortran 90 planned. Some additional overhead is introduced as Babel uses a C-based internal object representation (IOR) to provide the glue between different languages. In general, the overhead incurred by this scheme is roughly two subroutine calls – client language to IOR to server language. In some cases there is an additional overhead due to data conversions between languages. Clearly, when significant computation takes place in methods called via Babel, these overheads are not noticeable [8]. But care is required with smaller methods that are called a large number of times and involve little work, as is common in many communication systems.

3 Why Can't We Just Use Existing Message-Passing Codes as Is?

The PVM and MPI libraries have both existed for many years, in several incarnations each, and these libraries have been shown to work well in stand-alone applications as well as when integrated with other software systems and libraries. The libraries can each be applied as traditional static libraries or used as shared libraries. In a typical run-time environment, several processes can use the same library cooperatively, as each process is allocated its own address space, symbolic name space, and region of stack heap for use with local library data storage.

When using shared libraries, global variables in the libraries are replicated for each *process* that links against the shared library. As such there can be no interference among different processes whether using a common shared or static library.

However, in a component framework, where several different components running in the same process might utilize a single shared library, there is not necessarily such a guarantee of “non-interference.” The messaging libraries do not generally *share* well among multiple component instantiations in the same framework process. Several unexpected results can occur based on how the libraries are integrated into the component framework, and not all of these results are favorable depending on the desired effect.

For example, the following configurations are possible in a component framework:

1. Single messaging service for the whole framework & all components
2. Individual messaging service per component
3. Messaging system provided as a distinct, independent component

In Case (1) above, there is a single library linked into the framework for use by all component instances. Whether static or shared, this one instance of the library is used to service any or all messaging requests by the framework or components. This case effectively requires only that the framework be linked with the given messaging library and executed within any messaging system run-time environment. The messaging library becomes a built-in “service” of the framework that component instances can use as needed by simply invoking library methods as usual. Components intending to use this special service will possibly require some additional information from the framework, such as an MPI communicator.

However, because only a single library instance exists with one corresponding copy of all global variables, all components and the framework must share a single messaging identity. This may pose problems in cases where multi-threading is used¹, or where components in the same framework (process) need to exchange messages independently. This same potential problem exists in traditional (non-component) codes and has been handled in messaging libraries using the concept of “message context” to separate message spaces. So in general, the “framework service” alternative is sufficient only if components within the same framework process do *not* need to communicate with each other aside from via CCA-port-based method invocations (and if the framework and messaging library are compatible in their use of threads).

It should be noted that a variation of Case (1) can occur even when the framework does not include a common instance of the messaging library. If each component is compiled with its own *shared* copy of the library, an equivalent scenario is achieved. Because all component instances exist in the same physical

¹ Extant implementations of PVM and MPI were designed for use in a single-threaded process model, so support for multi-threading is uncommon; the use of PVM and MPI within a multi-threaded environment can be a precarious undertaking.

framework process, the operating system will provide a single shared library context for the whole process. So the components will share one common copy of the globals for the shared library, *even though they were compiled separately*, and the same rules for Case (1) apply. (Specific operating system details can influence this behavior.)

In Case (2), each component is compiled with its own *static* copy of the messaging library, and so has a unique library context and its own individual messaging service. Even though the component is itself a shared object, the static messaging library is encapsulated entirely within the component instance as a private, static entity. It should then be possible for each component to maintain its *own* identity within the messaging system, equivalent to utilizing different messaging contexts for each component. While intra-component communication might still be possible within a single framework process this would undermine the local port-based connections among components provided by CCA.

Scheme (2) provides the opportunity for independent messaging usage per component, with little potential for interference among components. Yet selecting such usage is subtle, and requires careful planning on the part of the component developer and end user. There is the possibility for confusion in expected behavior, and a simple compile-time error such as using a shared library instead of a static one could lead to program errors or even catastrophic failures. For example, if an existing MPI program were converted into a component and the build system was left primarily unchanged, then a static MPI library might still be linked into the component's shared object. Then if the user loaded the component into a framework with a built-in MPI service, expecting all local components to share a common MPI identity, the program would fail because each component would be secluded into its own independent communication space. Alternately, if the components were compiled with the shared MPI library, and a user expected them all to communicate independently the application would fail as all component instances would share the same messaging space. In fact, depending on the nature of the framework and the threading supported by the MPI implementation, in the worst case this oversight could cause a segmentation violation from simultaneous calls interfering with each other through an unexpectedly shared messaging system.

Case (3) also provides independent messaging contexts for each component, but provides more flexibility and less chance for inadvertent usage errors. Here, the messaging library is wrapped up and encapsulated as a component in the framework, at a peer level with other application components. The framework need not be aware of the messaging system (unless the framework itself needs it); rather the messaging system component gets instantiated into the framework like any other component. Other components wishing to invoke messaging functions simply create a "uses" port that can be connected to the "provides" port of the messaging component to access its interface methods. There could be several distinct messaging component instances in the same framework, whether identical or for different messaging libraries, so components could explicitly choose any compatible communication implementation. If a single messaging component is

utilized by all other components for their messaging, then the whole collection would constitute a single identity in the messaging environment. Alternately, each component could be connected to its own distinct instance of the messaging component to maintain a separate messaging identity. In this way, the functionality of both Cases (1) and (2) is encapsulated in Case (3).

Yet there is a distinct difference in the *usage* of Case (3), as opposed to Cases (1) and (2), involving the manner in which the methods in the messaging library interface are invoked. In Cases (1) and (2), components can make traditional invocations of the messaging library routines, the same as they would if they were not components in a parallel framework. However in Case (3), all calls to the messaging system must take place via the CCA port mechanism. This means that prior to making any such invocations, a port handle must be obtained for the connection to the messaging component. Then, each messaging invocation must use that port handle to dereference the method function pointer and invoke the method. For newly developed software components, this variation on standard library invocation requires minimal additional coding. But for existing or legacy software that has been wrapped up as components, a significant amount of re-coding could be necessary.

Therefore, the selection from among the above 3 cases depends not only on the desired functionality for the integration of the messaging system into the component framework, but also on the degree to which existing components need to be modified to utilize a component-based messaging solution. The next section discusses some practical issues regarding the use of messaging systems in component frameworks, and the subsequent section deals with specific issues of porting existing messaging codes.

4 Specialized PVM and MPI Component/Framework Features

When integrating a message-passing system into a component framework, there are several capabilities that must be provided for practical usage by components. These features allow components to discover what messaging systems are currently available in the framework, and to acquire handles or communicators for a given messaging system. There are also some “startup” issues associated with different messaging systems that need to be dealt with in integrating them with frameworks.

If the messaging system is wrapped up as a distinct component, then components wishing to utilize it need only check to see if their *uses* port has been connected. If there is a connection then the component can assume the messaging system is available for usage and can proceed. But if the messaging service is integrated into the framework as a special service, then some additional mechanism is required to verify the existence of a given messaging system. In this case, there must be some sort of query method, as part of the standard **Services** interface, that can check whether the framework supports the given service. Without such a verification, the component would have to hope and assume that the service is

available, potentially leading to a component failure if the service is *not* available and an unresolved messaging library symbol is invoked.

Further, once the existence of a given messaging library is verified, a component will likely need some additional information to actually *use* the service. In the case of MPI, the component may or may not need to invoke `MPI_init` to initialize the system, depending on the way in which MPI was integrated into the framework and whether `MPI_init` has already been invoked by another component or the framework. Precise conventions need to be defined for each possible integration case, so that component developers can reliably initialize the messaging system as needed. Once the messaging system is initialized, the component must next acquire a communicator for MPI. It may not be the case that a global `MPI_COMM_WORLD` constant is the correct value for its localized component usage. The framework messaging service, or stand-alone messaging component, must provide a method for obtaining the proper communicator. This communicator should likely provide a unique messaging context for each component. (These issues are not relevant with PVM, as the `pvm.mytid` function can be called repeatedly to initialize the system and obtain the local task ID for each component; multiple calls do not affect the messaging configuration. Similarly, standard PVM library calls can be used to generate a unique context for a component cohort.)

Another issue involves startup of the messaging system environment. In PVM, the virtual machine must be created *before* the framework or any components can be instantiated, but there are no restrictions on how the framework executable is started or how components are loaded. For MPI, the `mpirun` script must be used to execute the framework, so the number and location of parallel framework processes must be known at initial run time. For this reason, it is not feasible to start an arbitrary collection of framework processes first, and *then* instantiate some MPI components into the framework as a cohesive messaging service. (While it may be possible to implement some form of bootstrapping using new MPI-2 server features, this approach is a bit unwieldy.) To best handle this case, the framework must always be started using `mpirun`, to create the proper parallel MPI context, in case there is any possibility that a given MPI component will need to be instantiated later.

A related issue involves the combined use of both MPI and PVM in a single framework application. Given the above `mpirun` restriction for “MPI-compliant” frameworks, there are several options for overlaying a PVM virtual machine run-time environment onto the same framework processes. One possibility is to simply apply a priori knowledge of where the MPI processes will be started, and preemptively start a PVM virtual machine on those same hosts. Then the PVM and MPI universes will simply overlap. A more sophisticated approach could take advantage of PVM’s dynamic resource management features and the `pvm.addhosts` function to automatically build PVM on top of an MPI working set. If a special “PVM Piggyback” component were instantiated into an MPI-compliant framework, then it could query each parallel task for information about its local host and then construct an overlapping PVM virtual machine

on-the-fly. The “Rank 0” MPI task could start the master PVM daemon on the local host (using the `pvm_start_pvmd` function). Then the piggyback component could utilize the MPI collective communication functions to pass the other host’s name information to the master host for adding to the PVM virtual machine. This would work well assuming the host names can be readily obtained from appropriate system calls, and that the proper PVM configuration exists on each given machine.

5 Componentizing Existing Codes and Path of Least Resistance

As outlined in section 3, there are several options for incorporating message passing systems into a component environment, and therefore several choices for componentizing existing codes that use message-passing. While integrating the messaging system into the framework as a service may be the simplest choice, it limits the alternatives for components in utilizing the service and restricts the reusability of individual components. This approach also complicates framework development and maintenance.

The use of an individual messaging service per component (using static libraries) has the advantage of requiring little or no modification to the build process of each component. However its feasibility is contingent upon the runtime messaging system supporting the existence of multiple instances or identities in the same process address space. The likelihood of a system providing this support is questionable; further, the resulting behavior could vary dramatically between different operating systems.

The use of an independent component for messaging system functions provides the flexibility of allowing multiple component collections with distinct messaging needs to safely coexist within the same process address space. However, this choice requires significant code modifications as the API for invoking port services is different from direct library calls. While some automation could be deployed to facilitate the migration from library calls to corresponding component port calls, manual code inspection and modification will invariably be necessary to complete the transformation.

Another factor that affects the componentization of existing code is the use of inter-language binding through Babel. This can allow access to the messaging functionality independent of the source language of the existing code (assuming it is supported by Babel). But using Babel-generated interfaces mandates changing the names of the functions to fit into the SIDL class hierarchy, as well as modifying argument types to correspond to SIDL primitive types. Language specific features (e.g. arrays and pointers) need to be replaced with SIDL equivalents (i.e. SIDL arrays and objects). These changes can be hidden in the client component code by inserting thin macro wrappers around the client-side Babel code. Such wrapping could also be used to hide the object-oriented nature of Babel and reduce the amount of code modification required in the client components. Because Babel introduces some overhead to method invocations,

experimentation and performance analysis are needed to quantify this overhead in the context of components that use message-passing.

With respect to the level of effort required to encapsulate a messaging system in a component framework, the messaging component approach has the advantage of requiring minimal additional code. This additional code will primarily handle language interoperability issues that arise through the use of Babel, and provide the mapping from the object oriented approach inherent in the component model to the native design of the messaging libraries. An effort is currently underway at Oak Ridge National Laboratory to provide such encapsulation of the PVM libraries as components, and researchers at ANL are developing SIDL bindings for MPI. Such SIDL bindings are a major part of the effort of componentization. This work will provide a better understanding of the ramifications to runtime performance of messaging in a component framework, and will produce usable prototype message passing components for further testing.

Another approach to the componentization of message-passing systems is to develop a new set of abstract interfaces that cover the spectrum of messaging functionalities. This set of interfaces would cover areas such as point-to-point communications, collective operations, one-sided communications, dynamic resource discovery and management, events and notifications, I/O, etc. A component developer could implement all or a subset of these services, leaving the application designer to choose from several implementations based on application requirements. However, this approach would require the development of generalized interfaces to subsume the existing PVM and MPI standards, and these interfaces would be dramatically different (at the API level) from the existing libraries and usage. Yet these interfaces could be developed and explored by the scientific computing community as a long-term effort to enhance the development of next-generation scientific codes.

6 Conclusions

The use of component-based systems in high-performance computing environment poses new challenges that require new approaches to the deployment of message-passing infrastructure. The use of existing message-passing libraries “as is” poses significant architectural and functional problems. While some of these problems are common to any componentized library that might be shared by several components, the special problems of how to bootstrap a component-based application in a parallel/distributed environment are unique and dependent on the underlying messaging environment’s design. The incorporation of componentized message-passing libraries could require significant modification of application codes and likely introduces some runtime overhead, and it would be useful for the developers of messaging environments and component-based scientific software to begin a dialog aimed at producing a general, robust approach to these issues. Work has begun by the authors on a prototype componentized version of the PVM library; this prototype will be utilized to empirically demonstrate and analyze the functionality and performance impacts presented here.

References

- [1] CCA Forum home page. <http://www.cca-forum.org>. 261
- [2] Components @ LLNL: Babel. <http://www.llnl.gov/CASC/components/babel.html>. 263
- [3] Benjamin A. Allan, Robert C. Armstrong, Alicia P. Wolfe, Jaideep Ray, David E. Bernholdt, and James A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, in press. 261
- [4] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, , and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, 1998. 261
- [5] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidya Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994. 261
- [6] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*, volume 2 – The MPI-2 Extensions. MIT Press, September 1998. 261
- [7] Object Management Group. OMG's CORBA website. <http://www.corba.org>. 261
- [8] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2001. Also available at <http://www.llnl.gov/CASC/components/publications.html>. 263
- [9] V. Matena, M. Hapner, and B. Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. The Java Series. Addison-Wesley, 2000. 261
- [10] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *J. Supercomputing*, 10(2):169, 1996. 263
- [11] Boyana Norris, Satish Balay, Steve Benson, Lori Freitag, Paul Hovland, Lois McInnes, and Barry Smith. Parallel components for PDEs and optimization: Some issues and experiences. Technical Report ANL/MCS-P932-0202, Argonne National Laboratory, February 2002. Available via <http://www.mcs.anl.gov/cca/papers/p932.pdf>; under review as an invited paper in a special issue of *Parallel Computing on Advanced Programming Environments for Parallel and Distributed Computing*. 261
- [12] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, 1997. 261
- [13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*, volume 1 – The MPI Core. MIT Press, 2nd edition, September 1998. 261

On Benchmarking Collective MPI Operations

Thomas Worsch¹, Ralf Reussner², and Werner Augustin¹

¹ LIIN, Universität Karlsruhe, Germany
worsch@ira.uka.de

werner.augustin@aoeu.de

² DSTC, Monash University, Australia
rreussner@dstc.monash.edu.au

Abstract. This article concentrates on recent work on benchmarking collective operations with *SKaMPI*. The goal of the *SKaMPI* project is the creation of a database containing performance measurements of parallel computers in terms of MPI operations. These data support software developers in creating portable *and* fast programs. Existing algorithms for measuring the timing of collective operations are discussed and a new algorithm is presented, taking into account the differences of local clocks. Results of measurements on a Cray T3E/900 and an IBM RS 6000 SP are presented.

1 Introduction

The MPI standard defines a set of powerful collective operations useful for coordination and communication between many processes. Knowing the quality of the implementations of collective operations is of great interest for application programmers.

SKaMPI is the Special Karlsruher MPI-Benchmark [7]. *SKaMPI* measures the performance of MPI implementations. While other benchmarks (e.g., for the top-500 list) try to express the performance of a machine in one figure to ease the ranking of machines, *SKaMPI*'s primary goal is to support software developers. By providing detailed data about the performance of each MPI operation, a software developer can judge the consequences of design decisions regarding the performance of the system to be built (see [6] for a detailed discussion and further examples).

Until recently, *SKaMPI* measured the timing of collective operations with an approach relying on barrier synchronization. While this approach was state of the art when *SKaMPI* was designed in 1996, more recent work [2, 3, 4] shows that this approach has a systematic error in case of overlapping collective operations and by including the time spent for the additional barrier operation. A new approach to measure collective operations [2] is restricted to measurements of the `MPI_Bcast` operation and is relatively slow.

The contribution of this paper is a general approach to measure the performance of arbitrary collective operations in a relatively efficient way. The main results are taken from [1].

This article is organized as follows. In Section 2 we discuss existing approaches for defining and benchmarking the timing of collective operations. A new benchmark algorithm is described in Section 3 and it is compared to the algorithm previously used in *SKaMPI* in Section 4. Results from measurements on a Cray T3E/900 and an IBM RS 6000 SP are shown. Section 5 concludes and discusses future work.

2 Background

Benchmarking Collective Operations. Collective operations are of particular importance when writing message passing programs. The ability to perform complex coordination and communication patterns makes collective operations one of the few higher-level constructs in the otherwise relatively low-level (i.e., non-abstract, close to hardware) message-passing programming style. Hence, the MPI standard provides a large variety of collective operations and discusses their application for complex data exchange operations in detail [8]. Furthermore, they allow the use of optimized algorithms, which can make a complex collective operation faster than using semantically equivalent point-to-point operations in a straightforward manner.

Timing of Collective Operations and other Approaches. Since several processes are involved in a collective operation different definitions for the timing of collective operations are possible. It may be the time needed

1. for the collective operation at a designated process.
2. for the collective operation at the process which takes the longest time
3. between the first process starting and the last process finishing the collective operation.

The approach previously used in *SKaMPI* is a variation of the first alternative. We measure the time used by a collective operation on the root node plus the time needed by the root node to finish a subsequent `MPI_Barrier`. The barrier synchronization ensures that the collective operation has finished on every node, but it has the severe drawback that it can overlap with the collective operation to be measured. Therefore one cannot simply subtract the time used by a barrier alone to get the time used by the collective operation; and by using the same communication network the barrier operation can make things even worse and delay the measured operation. If one executes several measurements (as is usually done to get more reliable results), the barrier might not only overlap with the collective operation before, but also with the one following.

The algorithm for measuring collective operations presented in [2] is able to measure the timing of a collective operation according to definition 2. By changing the selection of the other node during repeated measurements, one gathers the times used by the collective operation for the root node to each other node. The result of the measurement is the maximum of the average time for the

broadcast to any single node. But what we actually want to know is the average of the maximum of the times. The exchange of maximizing and averaging is only permitted if the communication patterns used by the MPI implementation are always the same. Imagine that the MPI implementation alternates between two different implementations: the first one takes 1 time unit on node A and 5 on node B and the other one takes 5 time units on node A and 1 on node B. Because every operation takes 5 time units the result should be 5, though the average time for every node is 3 and the maximum of these times therefore would be 3 as well. The approach described above is also limited to collective operations, where a designated root node starts the operation (e.g., `MPI_Bcast`). But in many collective MPI operations, such as `MPI_Barrier`, `MPI_Gather` or `MPI_Alltoall`, there is no designated root node. In this paper we present a novel approach to overcome these limitations.

Problems of Benchmarking Collective Operations. Without a barrier synchronization after the measured collective operation, one cannot ensure that all processes have finished the collective operation. This problem also occurs when the maximum of the execution times on all participating processes is taken (definition 2 in the above list). To see why the algorithm should wait until the collective operation finished on *all* processes, consider the case of a broadcast operation which sends a message to all its recipients as if they were ordered sequentially. The message is sent from process 0 to process 1, from 1 to 2, from 2 to 3, and so forth. This results in a communication time increasing linearly with the number of processes involved. Although the root process probably finishes faster than in an algorithm with tree-like communication, in general the latter is preferable, of course.

As a result of this discussion, we see that definition 3 is most suitable to define the timing of a collective operation. It neither has the problems coming with the other definitions nor does it need a troublesome barrier synchronization after each measured routine.

3 A New Algorithm Based on Global Time

If definition 3 above is to be used then times taken on different processes have to be compared. But MPI does not guarantee that there is any relation between such local times.

A very elaborate and complex approach of clock synchronization is presented in [5]. This approach makes use of a log file, storing local times, which allows an off-line approximation of global times. Since our benchmark needs global times at run time and the creation of a log file in advance complicates the benchmark considerably, we preferred a simpler approach.

Augustin [1] suggests to use global time points which are obtained by approximating on each process i the offset o_i of the local time with respect to a global time. By definition let $o_0 = 0$, i.e. the local time of process 0 is the global time.

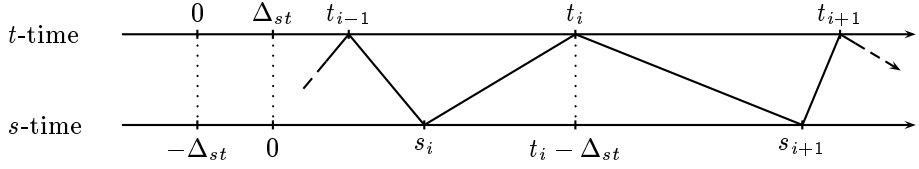


Fig. 1. Communication and timing principle for determining local time offsets

Computing the Difference between Two Local Times. Consider two processes named s and t in Figure 1. Their local times are represented by the two horizontal arrows. Let's call s -time the local time of s and t -time the local time of t . Let Δ_{st} denote the t -time corresponding to s -time 0.

Let the two processes send messages to each other in a ping-pong manner. A message leaving s at s -time s_i arrives at t at t -time t_i , and a message leaving t at t -time t_i arrives at s at s -time s_{i+1} .

Since t -time t_i corresponds to s -time $t_i - \Delta_{st}$ and a message cannot arrive before it has been sent, it is clear that $s_i < t_i - \Delta_{st} < s_{i+1}$. From this follow $\Delta_{st} < t_i - s_i$ and $\Delta_{st} > t_i - s_{i+1}$. If several messages i are sent, these inequations are true for all of them, and therefore $\max_i(t_i - s_{i+1}) < \Delta_{st} < \min_i(t_i - s_i)$. The times $t_i - s_i$ depend on the communication from s to t and the times $t_i - s_{i+1}$ on the communication from t to s . Minimizing $t_i - s_{i+1}$ and maximizing $t_i - s_i$ means to consider only the fastest communication. Under the assumption that the time for a fastest communication does not depend on the direction it is reasonable to approximate Δ_{st} as $\Delta_{st} \approx (\max_i(t_i - s_{i+1}) + \min_i(t_i - s_i))/2$.

A simplified sketch of the algorithm to compute Δ_{st} on process t is given in Algorithm 1. Simple ping-pong messages are used in order to quickly get a sequence of points in time for which their order is known.

Algorithm 1. Sketch of a non-optimized algorithm to compute Δ_{st} :

```

if ( $pid = s$ ) then
     $\Delta_{lb} \leftarrow -\infty$ ;  $\Delta_{ub} \leftarrow +\infty$ ;
    for  $i \leftarrow 0 \dots N$  do
         $s_{last} \leftarrow \text{MPI\_Wtime}()$ ;
         $\text{MPI\_Send}(\dots)$ ;
         $\text{MPI\_Recv}(\&t_{last}, \dots)$ ;
         $s_{now} \leftarrow \text{MPI\_Wtime}()$ ;
         $\Delta_{lb} \leftarrow \max(\Delta_{lb}, t_{last} - s_{now})$ ;
         $\Delta_{ub} \leftarrow \min(\Delta_{ub}, t_{last} - s_{last})$ ;
    od
     $\Delta \leftarrow (\Delta_{lb} + \Delta_{ub})/2$ ;

else /* ( $pid = t$ ) */
    for  $i \leftarrow 0 \dots N$  do
         $\text{MPI\_Recv}(\dots)$ ;
         $t_{last} \leftarrow \text{MPI\_Wtime}()$ ;
         $\text{MPI\_Send}(\&t_{last}, \dots)$ ;
    od

```

Experiments on several computing environments show that this gives a good approximation of the difference between the local times on different processes.

But the computed value is a good approximation only around the time when the measurements were done. There are machines where the local clocks are running at different speeds, i.e. the local time offsets change over time. They vary so much that after one hour (a time which a full *SKaMPI* run may need) the time offset computed at the beginning is not an adequate approximation any longer. It is therefore necessary to adjust the local time offsets every once in a while.

The Use of Global Time in *SKaMPI*. By definition the local time of process $s = 0$ in `MPI_COMM_WORLD` is the global time. For each $t \neq 0$ the algorithm sketched in Algorithm 1 is used to compute the offset o_t of t -time with respect to global time; process 0 sends the value o_t to process t . In the following we will denote global times by T (possibly with an index). Using its offset o_t each process t can easily translate global times to local times and vice versa.

In *SKaMPI* a *single measurement* is a single experiment to determine the time needed by an operation for a specific set of parameters, e.g. an `MPI_Bcast` for 8 processes with 64 kB message length. The outcome of a single measurement is a single value, e.g. $18.3 \mu\text{s}$. A single measurement of a collective operation is now roughly done as follows.

Algorithm 2. Using global time for benchmarking a collective operation:

1. Process 0 selects a time slot in the future characterized by T_{start} and T_{end} .
2. T_{start} and T_{end} are sent to all processes taking part in the operation.
3. At T_{start} each process calls the collective operation. If for some reason T_{start} has already passed on a processor this is an error.
4. Once the call returns on a process p , it checks whether the current global time T_p is less than T_{end} . If not, this is an error.
5. If on a process p an error has occurred, T_p is set to a value indicating it.
6. Using an `MPI_Gather` all T_p are collected on process 0 and evaluated:
 - If all T_p are valid, their maximum is the result of the single measurement.
 - If at least one T_p indicates an error, the results are discarded.

Optimizations. As mentioned above, the *SKaMPI* benchmark has to recompute the local time offsets from time to time. But the full computation for approximating the local time offsets (which is done when *SKaMPI* has been started) does take some time. If these computations are done too often too much time is wasted by *SKaMPI*. Therefore an adaptive updating of the local time offsets has been implemented. After each measurement (i.e. a sequence of single measurements needed to determine a value with a sufficiently small deviation) Algorithm 1 is started again, but will be stopped prematurely if the time for the used ping-pong messages is within 5% of the shortest time they needed at the initial, extensive synchronization run.

There are several problems concerning the length of time slots:

- In the beginning a reasonable length for the time slots is completely unknown. *SKaMPI* is supposed to run on many platforms so that not even the order of magnitude of the time for a collective operation is known.

- If the time slots are too short, too many single measurements exceed T_{end} and have to be discarded.
- If the time slots are too long, precious time is wasted.
- Not every exceeding of T_{end} should lead to a prolongation of the time slot. It may simply be due to a rare operating system event.
- If the length of time slots changes all processes have to be informed about that fact. This administrative overhead should be kept as small as possible.

In order to mitigate these problems *SKaMPI* uses the following approach. So-called *rounds* of single measurements are used. A round is characterized by the number n of single measurements, the length ℓ of the time slots for all of them and the global starting time T_{start} of the first measurement. This implies that for $i = 0, \dots, n - 1$ the time slot of the i th single measurement of the round is $(T_{start} + i\ell, T_{start} + (i+1)\ell)$. After a whole round is completed the measured times are collected and evaluated by process 0. If too many single measurements were invalid (because it took too long on a process or because a process could not start in time) the length of time slots is doubled and (if necessary) further increased so that all measurements of the previous round could have fitted 'comfortably' in their slots (i.e. one measurement would have used about $\frac{2}{3}$ of a slot).

If more than 50 % of all invalid measurements occur in a row, this is taken as a hint that possibly only one single event (e.g. an operating system interrupt) has delayed a whole sequence of measurements. Therefore the number of measurements in a round is reduced, but not below 4. To get a good estimate of the magnitude of the results (and therefore of the length of the time slots), the initial length is 0 and the first round is limited to 4 single measurements.

The numbers above have been found empirically and seem to give a good balance between administrative overhead on one hand and overhead due to invalid measurements on the other hand. The overhead of this new approach has been measured [1]. The results show that the new version of *SKaMPI* usually needs about a factor 2 to 2.4 longer than the old version. This is a significant amount, but it seems to be quite difficult to further improve the algorithm and its parameters in a way which is independent of a particular machine.

4 Results

We will now present benchmark results obtained on the Cray T3E of the HLRS Stuttgart (T3E for short) and on the IBM SP-256 of the University of Karlsruhe (SP for short). As examples `MPI_Barrier`, `MPI_Alltoall` and a self-written collective waiting operation are used. The latter was implemented in two variants, one where process i waits $(i + 1) \cdot 100$ ns ("WaitPattern-up") and one where process i waits $(n - i) \cdot 100$ ns where n is the size of the communicator ("WaitPattern-down"). If all processes start this operation at the same (global) time then the time needed for it should always be $n \cdot 100$ ns. Figure 2 shows the results.

A comparison of the results e.g. for WaitPattern-up clearly shows the difference between the old and the new measurement algorithm. The influence of the `MPI_Barrier` on the times measured is obvious. It is also interesting that using

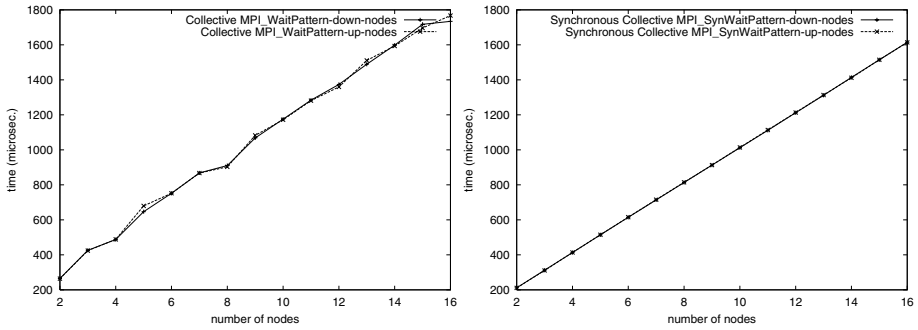


Fig. 2. Waiting operation measured on the SP with old (left) and new (right) *SKaMPI*

the old barrier method the measurements for `WaitPattern-up` and `WaitPattern-down` differ. This is surprising and demonstrates that it is not possible to correct the result by simply subtracting the time for an `MPI_Barrier` operation.

Figure 3 shows very interesting measurements of `MPI_Barrier`. The old method of measurement results in a sequence of two `MPI_Barrier` operations. As can be seen in the left part of Figure 3 for the IBM SP, this does in general *not* result in twice the time of one `MPI_Barrier`. This is due to the fact that the two operations may partially overlap. The figures for the T3E not only show a low latency and a more regular pattern, but that the times differ by a factor of 2, indicating that there is no overlap between the two operations.

Probably the most surprising result is shown in Fig. 4. An `MPI_Alltoall` was measured with relatively large messages (64kB) so that the time for a single `MPI_Barrier` “should” not have any noticeable effect on the overall time. But the influence is obvious.

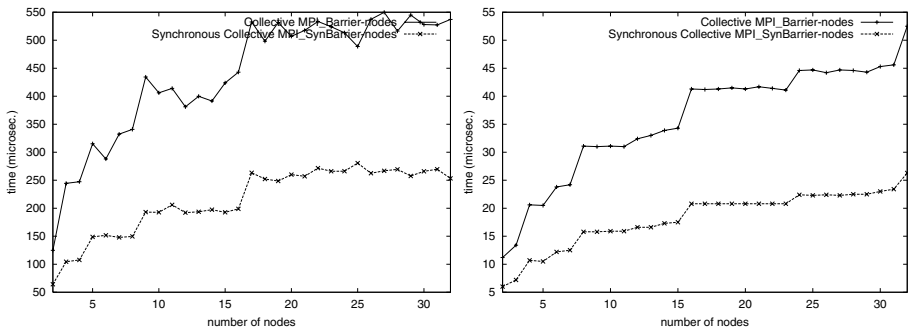


Fig. 3. `MPI_Barrier` measured with both *SKaMPI* versions on an SP (left) and a T3E (right). Graphs labeled “synchronous” corresponds to the new version

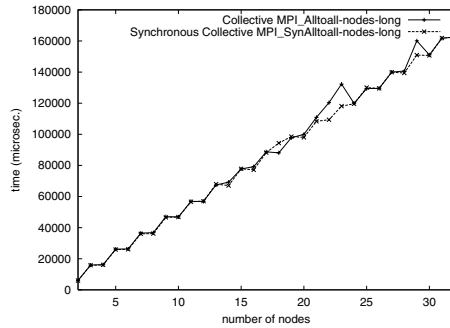


Fig. 4. `MPI_Alltoall` with 64kB messages with both *SKaMPI* versions on an IBM SP

5 Conclusions and Future Work in the *SKaMPI* Project

We have presented a novel approach for measuring the performance of collective operations. It relies on approximations of the time differences between local clocks. This information is used to start a collective operation simultaneously on all processes. The measurements of time differences are controlled by an adaptive validation mechanism. The new approach is compared to the previous approach used in *SKaMPI*. Measurements results on a Cray T3E and IBM SP are shown for a collective waiting operation and `MPI_Barrier`. Similar results for e.g. `MPI_Bcast` and `MPI_Gather` can be found in [1]. Of course, the differences are most notable for short messages and become less significant for long messages.

Besides improved results for standard benchmarks our new approach is the basis for investigations which previously were impossible. One can now analyze the real ending times of all processes of a collective operation started simultaneously and their “robustness” under varying starting times for different processes. As an extension we also plan to include measurements for more complicated communication patterns and common combinations of communication operations.

References

- [1] W. Augustin. Neue Messverfahren für kollektive Operationen. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, December 2001. [271](#), [273](#), [276](#), [278](#)
- [2] B.R. de Supinski and N.T. Karonis. Accurately measuring MPI broadcasts in a computational grid. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, August 2000. IEEE. [271](#), [272](#)
- [3] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In J. J. Dongarra, E. Luque, and T. Margalef (eds.) *EuroPVM/MPI, Barcelona, Spain, Sep. 26–29, 1999*, LNCS 1697, p. 11–18. Springer, 1999. [271](#)
- [4] R. Hempel. Basic message passing benchmarks, methodology and pitfalls, Sep. 1999. SPEC Workshop (www.hlrs.de/mpi/b_eff/hempel_wuppertal.ppt). [271](#)

- [5] R. Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen*. Dissertation, Univ. Stuttgart, March 2000. 273
- [6] R. H. Reussner and G. T. Hunzelmann. Achieving Performance Portability with SKaMPI for High-Performance MPI Programs. In V. N. Alexandrov et al. (eds.) *Proc. ICCS, Part II, San Francisco*, LNCS 2074, pp. 841–850. Springer, May 2001. 271
- [7] R. H. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In V. Alexandrov and J. J. Dongarra (eds.) *EuroPVM/MPI, Liverpool, Sep. 7–9, 1998*, LNCS 1497, p. 52–59. Springer, 1998. 271
- [8] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1. MIT Press, Cambridge, MA, 2nd edition, 1998. 272

Building Library Components that Can Use Any MPI Implementation

William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL
gropp@mcs.anl.gov
<http://www.mcs.anl.gov/~gropp>

Abstract. The Message Passing Interface (MPI) standard for programming parallel computers is widely used for building both programs and libraries. Two of the strengths of MPI are its support for libraries and the existence of multiple implementations on many platforms. These two strengths conflict, however, when an application wants to use libraries built with different MPI implementations. This paper describes several solutions to this problem, based on minor changes to the API. These solutions also suggest design considerations for other standards, particularly those that expect to have multiple implementations and to be used in concert with other libraries.

The MPI standard [1, 2] has been very successful (see, for example, [3]). Multiple implementations of MPI exist for most parallel computers [4], including vendor-optimized versions and several freely-available versions. In addition, MPI provides support for constructing parallel libraries. When an application wants to use routines from several different parallel libraries, however, it must ensure that each library was built with the *same* implementation of MPI. The reason rests with the lack of detailed specification in the header files `mpi.h` for C and C++, `mpif.h` for Fortran 77, or the MPI module for Fortran 90. Because the goals of MPI included high performance, the MPI standard gives the implementor wide latitude in the specification of many of the datatypes and constants used in an MPI program. For each individual MPI program, this lack of detailed specification in the header file causes no problems; few users are even aware that the specific value of, for example, `MPI_ANY_SOURCE` is not specified by the standard. Only the names are specified by the standard.

A problem does arise, however, in building an application from multiple libraries. The user must either mandate that a specific MPI implementation be used for all components or that all libraries be built with all MPI implementations. Neither approach is adequate; third-party MPI libraries (such as those created by ISVs) may be available only for specific MPI implementations, the building and testing each library for each MPI implementation are both time-consuming and difficult to manage. In addition, the risk of picking the wrong version of a library is high, leading to problems that are difficult to diagnose.

Furthermore, while building each library with each version of MPI is possible, this solution doesn't scale to other APIs that have multiple implementations. This is not a hypothetical problem. At least one ISV has already experienced this problem, as has any site that has installed MPICH, LAM/MPI, or MPICH-GM. Thus, a solution that allows an application to use *any* implementation of MPI (and, using similar techniques, other libraries) is needed.

At least two solutions to this problem are feasible. The first is completely generic. It effectively wraps all MPI routines and objects with new versions that work with any MPI implementation by deferring the choice of a specific implementation to link- or even run-time if dynamically-linked libraries are used. For this approach, one can use new tools for building language-specific interfaces from language-independent descriptions of components. The second approach may be applied to those MPI implementations whose opaque objects have the same size (e.g., `MPI_Request`s are four bytes in all implementations). Neither solution is perfect; both require some changes to the source code of an existing component that already uses MPI. In many cases, however, these changes can be automated, making their use almost transparent.

This paper reviews in Section 1 the MPI values and objects that are defined in the `mpi.h` header file. In Section 2, the paper describes the issues in creating a generic `mpi.h` for the C binding of MPI, with some comments about C++ and Fortran. Section 3 provides an example that shows the same object library used by two different popular MPI implementations.

The approach described here does not address the issues of performance portability, particularly the problem of choosing the MPI routines with the best performance. However, this approach does allow libraries to use the conventional approach of encapsulating performance-critical operations in separate routines and then using runtime configuration variables to choose the best approach. Performance measurement tools such as SkaMPI [6] can be used to help identify the appropriate choice for a particular MPI implementation.

1 Brief Overview of MPI

The MPI specification itself is language independent. The MPI-1 standard specifies bindings to C and Fortran (then Fortran 77); the MPI-2 standard added bindings for C++ and Fortran 90. MPI programs are required to include the appropriate header file (or module for Fortran 90); for C and C++, this is `mpi.h`. The file includes: typedefs for MPI opaque objects, definitions of compile-time constants, definitions of link-time constants (see below), and function prototypes. The implementation, however, is given wide latitude in the definition of the named objects. For example, the following definitions for the MPI object used to describe data layouts (`MPI_Datatype`) are used by four common implementations:

| Implementation | Datatype Definition |
|----------------|---|
| IBM | <code>typedef int MPI_Datatype;</code> |
| LAM | <code>typedef struct _dtype *MPI_Datatype;</code> |
| MPICH | <code>typedef int MPI_Datatype;</code> |
| SGI | <code>typedef unsigned int MPI_Datatype;</code> |

While no implementation appears to use a `struct` rather than an `int` or pointer, nothing in the MPI standard precludes doing so.

The MPI standard also requires each routine to be available with both MPI and PMPI prefixes. For example, both `MPI_Send` and `PMPI_Send` are required. This is called the profiling interface. The intent of the PMPI versions is to allow a tool to replace the implementations of the MPI routines with code that provides added functionality (such as collecting performance data); the routine can then use the PMPI version to perform the MPI operation. This is a powerful feature that can be exploited in many applications and tools, such as the performance visualization tools Jumpshot [7] and Vampir [5].

One solution, mentioned above, creates a new binding for MPI that specifies all of the values and types. A “user” implementation of this binding could make use of the profiling interface. In order to use this new binding, however, applications must be rewritten. This solution will be described in another paper.

2 A Generic MPI Header

An alternative solution to a new MPI binding is to find a common version of the header file `mpi.h` that can be used by multiple implementations. This solution is less general than the generic binding but has the advantage of requiring fewer changes to existing codes. The approach here is to replace compile-time constants with runtime constants by finding a common data representation for MPI objects. There are some drawbacks in this approach that are discussed below. However, the advantages are that it requires few changes to existing code and most operations directly call the specific MPI routines (with no overhead).

The approach is as follows: Define a new header file that replaces most compile-time values with runtime values. These values are then set when the initialization of MPI takes place. In most cases, this can be accomplished at link time by selecting a particular library, described below, that contains the necessary symbols.

The header file `mpi.h` contains the following items that must be handled:

Compile-time values. These include the constants defining error classes (e.g., `MPI_ERR_TRUNCATE`) and special values for parameters (e.g., `MPI_ANY_SOURCE`). These can be replaced with runtime values that are initialized by `GMPI_Init`.

A special case are the null objects such as `MPI_COMM_NULL`. Most implementations define all null objects the same way (either zero or -1). The MPICH-2 implementation, however, encodes the object type in each object handle, including the null object handles. Thus, these terms must cannot be defined at compiletime.

Compile-time values used in declarations. These include constants defining maximum string sizes (e.g., `MPI_MAX_ERROR_STRING`). In many cases, these can be replaced by either the maximum or the minimum over the supported implementations, for example,

| Implementation | Size of <code>MPI_MAX_ERROR_STRING</code> |
|----------------|---|
| SGI | 256 |
| IBM | 128 |
| LAM | 256 |
| MPICH | 512 |

Defining `MPI_MAX_ERROR_STRING` as 512 is adequate for all of these MPI-1 implementations, since this value is used only to declare strings that are used as output parameters. Other values describe the sizes of input arrays, such as `MPI_MAX_INFO_KEY`. In this case, the minimum value should be used. While this might seem like a limitation, a truly portable code would need to limit itself to the minimum value used in any supported MPI implementation, so this is not a limitation in practice.

Init-time constants. MPI defines a number of items that are constant between `MPI_Init` and `MPI_Finalize`, rather than compile-time constants. The pre-defined MPI datatypes such as `MPI_INT` belong to this category. For most users, the difference isn't apparent; it comes up only when users try to use one of these in a place where compile-time constants are required by the language (such as case labels in a switch statement). Init-time constants are actually easier to implement than compile-time constants because it is correct to define them as values that are initialized at run time. Fortunately, in most implementations, these are implemented either as compile-time constants or link-time constants (such as addresses of structures) and require no code to be executed to initialize them. Thus, we do not need a special `MPI_Init` routine to handle this case.

Opaque objects. These are both the easiest and the most difficult. In practice, most implementations define these as objects of a particular size, and further, most implementations choose to either `ints` or pointers to represent these objects. On many platforms, `ints` and pointers are the same size. For such platforms, we can simply pick one form (such as `int`) and use that.

If the size of these objects is different among different implementations, then there is no easy solution. We cannot pick the largest size because some MPI operations use arrays of opaque objects (e.g., `MPI_Waitsome` or `MPI_Type_struct`). The solution described here does not handle this case, though an approach that follows the technique used for `MPI_Status`, shown below, may be used.

One additional complication is the handle conversion functions introduced in MPI-2. These convert between the C and Fortran representations of the handles. For many implementations, these are simply cast operations, and the MPI standard allows them to be implemented as macros. However, for greatest flexibility, the generic header described here implements them as functions, permitting more complex implementations.

Defined objects (status). The most difficult case is `MPI_Status`. MPI defines this as a `struct` with some defined members such as `MPI_TAG`. An implementation is allowed to add its own members to this structure. Hence, the size of `MPI_Status` and the layout of the members may be (and is) different in each implementation. The only solution to this problem is to provide routines to allocate and free `MPI_Status` objects and to provide separate routines to access all of the elements. For example, instead of

```
MPI_Status status;
...
MPI_Recv( ..., &status );
if (status.MPI_TAG == 10 || status.MPI_SOURCE == 3) ...
```

one must use special routines such as

```
MPI_Status *status_p = GMPI_Status_create(1);
MPI_Recv( ..., status_p );
if (GMPI_Status_get_tag( status_p ) == 10 ||
    GMPI_Status_get_source( status_p )) ...
GMPI_Status_free( status_p, 1 );
```

Fortunately, many MPI programs don't need or use `status`. These programs should use the MPI-2 values `MPI_STATUS_NULL` or `MPI_STATUSES_NULL`.

Defined pointers. MPI also defines a few constant pointers such as `MPI_BOTTOM` and `MPI_STATUS_NULL`. For many implementations, these are just `(void *)0`. Like the most of the other constants, these can be set at initialization time.

`MPI_STATUS_NULL` is a special case. This is not part of MPI-1 but was added to MPI-2. If an MPI implementation does not define it, a dummy `MPI_Status` may be used instead. This will not work for the corresponding `MPI_STATUSES_NULL`, which is used for arguments that require an array of `MPI_Status`, but is sufficient for most uses.

The complete list of functions that must be used for accessing information in `MPI_Status` is as follows:

```
/* Create and free one or more MPI_Status objects */
MPI_Status *GMPI_Status_create( int n );
void GMPI_Status_free( MPI_Status *p );
/* Access the fields in MPI_Status */
int GMPI_Status_get_tag( MPI_Status *p, int idx );
int GMPI_Status_get_source( MPI_Status *p, int idx );
int GMPI_Status_get_error( MPI_Status *p, int idx );
int GMPI_Status_get_count( MPI_Status *p, MPI_Datatype dtype,
                           int idx, int *count );
```

For MPI-2, it is also necessary to add

```
void GMPI_Status_set_error( MPI_Status *p, int idx, int errcode );
```

To reduce the number of function calls, one could also define a routine that returns the tag, source, and count, given a status array and index. The routines shown above are all that are required, however, and permit simple, automated replacement in MPI programs.

These routines, as well as definitions of the runtime values of the various compile- and init-time constants, are compiled into a library with the name `libgmpit \textit{name}` , where *name* is the name of the MPI implementation. For example, a cluster may have `libgmpitompich` and `libgmpitolam`. The definitions of the runtime values are extracted from the `mpi.h` header of each supported MPI implementation; special tools have been developed to automate much of this process.

Most C++ and Fortran bindings for MPI are based on wrappers around the C routines that implementations use to implement MPI. Thus, for C++ and Fortran, a variation of the “generic component” or new MPI binding mentioned above can be used. For those objects and items that are defined as constants, the same approach of using variables can be used.

Fortran shares the same issues about `MPI_Status` as C does. For Fortran 90, we can use a similar solution, returning a dynamically allocated array of `MPI_Status` elements and providing a similar set of functions to access the elements of `MPI_Status`.

3 Using the Generic Header

Applications are compiled in the same way as other MPI programs, using the header file `mpi.h`. Linking is almost the same, except one additional library is needed: the implementation of the GMPI routines in terms of a particular MPI implementation. For example, consider an application that uses GMPI and a library component that is also built with GMPI. The link line looks something like the following:

```
# Independent of MPI implementation (generic mpi.h in /usr/local/gmpi)
cc -c myprog.c -I/usr/local/gmpi/include
cc -c mylib.c -I/usr/local/gmpi/include
ar cr libmylib.a mylib.o
ranlib libmylib.a
# For MPICH
/usr/local/mpich/bin/mpicc -o myprog myprog.o -lmylib \
    -L/usr/local/gmpi/lib -lgmpitompich
# For LAM/MPI
/usr/local/lammpi/bin/mpicc -o myprog myprog.o -lmylib \
    -L/usr/local/gmpi/lib -lgmpitolam
```

With this approach, only one compiled version of each library or object file is required. In addition, for each MPI implementation, a single library implementing GMPI in terms of that implementation is required.

As an illustration, shown below is a simple library that can be used, in compiled form, with two different MPI implementations on a Beowulf cluster.

This simple library provides two versions of a numerical inner product: one that uses `MPI_Allreduce` and is fast and one that preserves evaluation order for the allreduce operation (unlike real numbers, floating-point arithmetic is not associative, so the order of evaluation can affect the final result), at a cost in performance. The code for the library routine is as follows:

```
#include "mpi.h"

double parallel_dot( const double u[], const double v[], int n,
                    int ordered, MPI_Comm comm )
{
    int    rank, size, i;
    double temp = 0, result;

    if (ordered) {
        MPI_Comm tempcomm;
        /* A good implementation would cache the duplicated communicator */
        MPI_Comm_dup( comm, &tempcomm );
        MPI_Comm_rank( tempcomm, &rank );
        MPI_Comm_size( tempcomm, &size );
        if (rank != 0)
            MPI_Recv( &temp, 1, MPI_DOUBLE, rank-1, 0, tempcomm,
                     MPI_STATUS_NULL );

        for (i=0; i<n; i++)
            temp += u[i] * v[i];
        if (rank != size-1)
            MPI_Send( &temp, 1, MPI_DOUBLE, rank+1, 0, tempcomm );
        MPI_Bcast( &temp, 1, MPI_DOUBLE, size-1, tempcomm );
        MPI_Comm_free( &tempcomm );
        result = temp;
    }
    else {
        for (i=0; i<n; i++)
            temp += u[i] * v[i];
        MPI_Allreduce( &temp, &result, 1, MPI_DOUBLE, MPI_SUM, comm );
    }
    return result;
}
```

This code uses the communicator that is passed into the routine. The following shell commands show how easy it is to build this library so that it may be used by either MPICH or LAM/MPI. The main program is in `myprog` and includes the call to `MPI_Init` and `MPI_Finalize`.

```
% cc -c -I/usr/local/gmpi/include dot.c
% ar cr libmydot.a dot.o
% ranlib libmydot.a
% cc -c -I/usr/local/gmpi/include myprog.c
% /usr/local/mpich/bin/mpicc -o myprog myprog.o \
```

```

        -lmydot -L/usr/local/gmpi/lib -lgmpitompich
% /usr/local/mpich/bin/mpirun -np 4 myprog
% /usr/local/lammpi/bin/mpicc -o myproc myprog.o \
        -lmydot -L/usr/local/gmpi/lib -lgmpitolam
% /usr/local/lammpi/bin/mpirun -np 4 myprog

```

This example has been run as shown with an implementation of the libraries `libgmpitompich` and `libgmpitolam` built for an IA32 cluster.

4 Conclusion

This paper has outlined an approach that allows libraries and object files to use several different MPI implementations without requiring recompilation and without requiring significant changes in the source for those components. In fact, some applications will require no changes at all. An implementation of this approach for MPICH and LAM/MPI on IA32 platforms has been constructed and demonstrated. In addition, source code transformation tools have been developed that aid in constructing the `libgmpitoname` library for other MPI implementations. These tools and libraries will be available at <http://www.mcs.anl.gov mpi/tools/genericmpi>.

Acknowledgments

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

References

- [1] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994. 280
- [2] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998. 280
- [3] List of papers that use MPI. <http://www.mcs.anl.gov mpi/mpiarticles>. 280
- [4] MPI implementations. <http://www.mcs.anl.gov mpi/implementations.html>. 280
- [5] Vampir 2.0 – Visualization and Analysis of MPI Programs. <http://www.pallas.de/pages/vampir.htm>. 282
- [6] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998. 281
- [7] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999. 282

Stampi-I/O: A Flexible Parallel-I/O Library for Heterogeneous Computing Environment

Yuichi Tsujita¹, Toshiyuki Imamura^{1,2},
Hiroshi Takemiya³, and Nobuhiro Yamagishi¹

¹ Center for Promotion of Computational Science and Engineering

Japan Atomic Energy Research Institute

6-9-3 Higashi-Ueno, Taito-ku, Tokyo 110-0015, Japan

{tsujita, imamura, yama}@koma.jaeri.go.jp

² High Performance Computing Center Stuttgart

Allmandring 30, D-70550 Stuttgart, Germany

³ Hitachi Tohoku Software, Ltd.

2-16-10 Honcho, Aoba-ku, Sendai, Miyagi 980-0014, Japan

Abstract. An MPI-2 based parallel-I/O library, Stampi-I/O, has been developed using flexible communication infrastructure. In Stampi-I/O almost all MPI-I/O functions have been implemented. We can execute these functions using both local and remote I/O operations with the same application program interface (API) based on MPI-2. In I/O operations using Stampi-I/O, users need not handle any differences in the communication mechanism of computers. We have evaluated performance for primitive functions in Stampi-I/O. Through this test, sufficient performance has been achieved and effectiveness of our flexible implementation has been confirmed.

1 Introduction

Data-intensive applications require huge amounts of memories, disks in addition to computation power. A parallel computer, however, has a limit on these resources due to physical constraints. One of the solutions to overcome this limit is to make a large scale computing environment by connecting distributed computers. This would enable the large scale distributed parallel computation.

Although MPI [1, 2] is the de facto standard in distributed parallel computation, there is no vendor supplied MPI library which is available among multiple platforms. To realize MPI communication on such an environment, we have designed and developed a distributed MPI communication library, Stampi [3]. It enables us large scale distributed parallel computation without awareness of underlying communication mechanisms.

Recently, several kinds of parallel I/O systems have been developed and used for effective handling of huge data. Parallel I/O functions named MPI-I/O was defined in MPI-2, and they are designed to handle such huge data effectively. But there are no vendor supplied libraries which are designed to function across multiple computers, several kinds of libraries have been developed

such as ROMIO [4] and PACX-MPI PIO [5]. ROMIO is a well-known library included in MPICH [6]. In ROMIO, many kinds of parallel-I/O systems are available through ADIO [7]. When we use it on unsupported I/O systems, we need to introduce additional interface libraries.

To realize a flexible parallel-I/O library, we have designed and developed a distributed parallel I/O library, Stampi-I/O, using a flexible communication infrastructure. In our concept, we do not make interface libraries for several kinds of I/O systems like ADIO. Stampi-I/O has been designed to work on any computer where a vendor supplied MPI library is supported. We can execute MPI-I/O functions using both local and remote I/O operations within the same API.

In this paper, outline, architecture and preliminary results of Stampi-I/O are described.

2 Stampi-I/O

Stampi-I/O has been developed as a flexible distributed MPI-I/O library in Stampi. The operation of Stampi-I/O is realized with inter-machine data transfer with flexible communication infrastructure of Stampi and I/O operation with a vendor supplied MPI-I/O library on a target machine. Stampi-I/O has the following features;

1. Flexible mechanism in local and remote I/O operations,
2. Flexible communication mechanism using router processes,
3. Remote I/O operations using MPI-I/O processes and
4. Flexible adaptability which is independent of architecture of computers.

We can use MPI-I/O functions using Stampi-I/O without consideration of differences in communication mechanism and I/O system. Architectural view of Stampi-I/O is shown in Fig. 1. A Stampi library including a Stampi-I/O library is linked to user processes, a vendor supplied MPI library and router processes bridging the gap between user processes on the different machines. Those router processes are created on IP-reachable nodes. In the following subsections, we describe the details of Stampi-I/O.

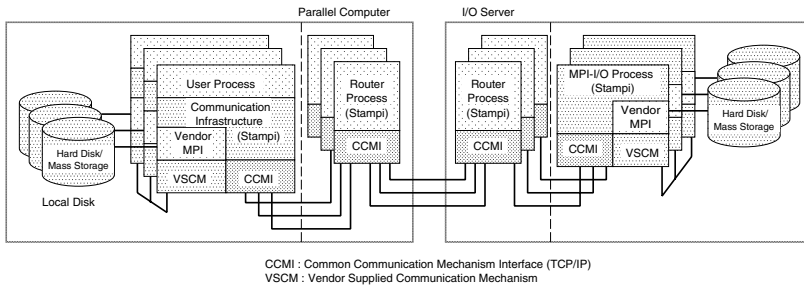


Fig. 1. Architectural view of Stampi-I/O

2.1 Flexible Mechanism in Local and Remote I/O Operations

In the first step, user processes are executed on a parallel computer by a Stampi start-up command as shown in Fig. 1. When user processes call `MPI_File_open`, a communication infrastructure library is called initially. This intermediate library has been implemented to have inter-operability in both local and remote I/O operations with the same API from user processes.

In the next step, the Stampi-I/O library selects either local or remote I/O methods. When `MPI_INFO_NULL` is set in `MPI_File_open`, the local I/O method is selected. In this case, a vendor supplied MPI-I/O library is directly called from the communication infrastructure library. Afterward the I/O operation is carried out on a local machine. Intra-machine data transfer among user processes is realized by VSCM (vendor supplied communication mechanism). On the other hand, when target host name, user-ID, working directory name, etc. are set in an `info` object by `MPI_Info_set` and the `info` is used in `MPI_File_open`, the remote I/O method is selected. Remote I/O operation is carried out through CCMI (common communication mechanism interface) layer via the router and MPI-I/O processes. In Stampi-I/O, TCP/IP is used in the CCMI layer.

2.2 Flexible Communication Mechanism Using Router Processes

Let us examine two cases of inter-machine data transfer; (a) only dedicated nodes can communicate with external computers, and (b) every nodes is fully connected. To realize inter-machine data transfer on every nodes in the case of (a), a router process has been introduced in Stampi-I/O as shown in Fig. 1. On the contrary, no router process is created usually in the case of (b). In this case, user processes directly communicate externally. The mode of operation is selected dynamically according to communication mechanism of computers.

2.3 Remote I/O Operations Using MPI-I/O Processes

Stampi-I/O supports both interactive and batch mode. Here, we focus on I/O operations with batch mode as shown in Fig. 2. Firstly, user processes are initiated by a Stampi start-up command (starter). When `MPI_File_open` is called by user processes, a starter process is initiated on a remote machine if a target host is a remote machine. Then the starter on the remote machine generates a script file which is submitted to a batch queue system according to specified queue class in an `info` object. Secondly, the script file then invokes an additional starter process that itself invokes an MPI-I/O process and a router process if it is required. After network connection has been established among user processes, router processes and an MPI-I/O process, I/O requests from user processes are translated into a message to the MPI-I/O process. Finally, an MPI-I/O process issues I/O calls on the remote machine using a vendor supplied MPI-I/O library according to the message.

In addition, multiple MPI-I/O processes can be created on a remote machine for performance improvement if a user requests to use this functionality in user

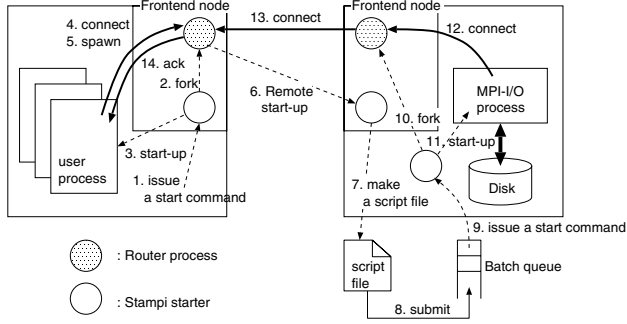


Fig. 2. Dynamic creation mechanism of MPI-IO process

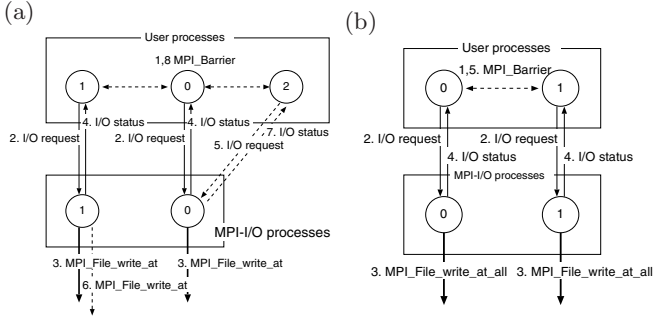


Fig. 3. Functional mechanism of `MPI_File_write_at_all` with multiple MPI-I/O processes. The number of user processes is greater than that of MPI-I/O processes in the case of (a) and both numbers are the same in the case of (b)

programs. The number of MPI-I/O processes is also variable up to the number of user processes.

Next, we focus the implementation of MPI-I/O functions. As an example, we explain the functional mechanism of `MPI_File_write_at_all` shown in Fig. 3. Network connections between user processes and MPI-I/O processes are established in round-robin manner according to rank-ID. Data transfer between both processes is carried out via non-blocking connection using TCP/IP. In the beginning, `MPI_Barrier` is issued among user processes for synchronization. In the case of Fig. 3 (a), the number of user processes is greater than that of MPI-I/O processes. I/O requests of user processes of rank 0 and 1 are sent to each corresponding MPI-I/O process at first. Although a requested I/O function among user processes is `MPI_File_write_at_all`, each MPI-I/O process operates vendor supplied `MPI_File_write_at` sequentially according to its own rank-ID. In the next phase, an MPI-I/O process of rank 0 operates the same function according to I/O request from a user process of rank 2. While in the case of Fig. 3 (b), where the numbers of user processes and MPI-I/O processes are

the same, the same function, `MPI_File_write_at_all`, is used using a vendor supplied MPI-I/O library among MPI-I/O processes. After the I/O operations complete, `MPI_Barrier` is issued among user processes in both cases.

2.4 Flexible Adaptability Which is Independent of Architecture of Computers

Inter-machine data transfer requires us to handle byte-order, floating-point format and so on, for each computer architecture. To realize flexible and user-friendly inter-machine data transfer, **external32** format defined in MPI-2 is adopted in our flexible communication infrastructure. The **external32** format is also adopted in I/O operations to have inter-operability across multiple platforms.

3 Performance Measurement of Stampi-I/O

Performance measurement tests using Stampi-I/O were carried out using a Fujitsu VPP5000 and a Hitachi SR8000. Hereafter, they are denoted as VPP and SR, respectively. They are interconnected via gigabit Ethernet based LAN. In this test, peak performance of collective blocking I/O operations was measured using `MPI_File_write_at_all` and `MPI_File_read_at_all` for both local and remote operations. In addition, performance of remote operations using multiple MPI-I/O processes was measured. In this test, data was distributed among user processes equally. In the following subsections, we describe data size as the whole data size among user processes. Details of performance measurement method and preliminary results are discussed in the following subsections.

3.1 Performance of Local I/O Operations

Firstly, performance of local I/O operations with Stampi-I/O was measured using a vendor supplied MPI-I/O library [8] on the VPP. Performance results of local I/O operations are shown in Table 1. In this table, **np** denotes the number of user processes. From this table, performance advantage for two and four user processes was found compared with performance for a single user process.

In the vendor supplied MPI-I/O library on the VPP, disk cache mechanism is implemented for performance improvement. In this mechanism, the number of cache blocks is 256 and the size of each block is 64 KByte. Thus, total size of the cache is 16 MByte. We considered that performance improvement was realized with the help of the disk cache mechanism in the case of **np = 2, 4**. Besides it is considered that performance degradation for over 16 Mbyte was caused by inefficiency of the mechanism when the cache was almost full with data.

In addition, we have also measured performance in the vendor supplied execution method. At most 6 % degradation was observed in the Stampi supplied execution method compared with the vendor supplied execution method. Thus Stampi supplied execution method does not have significant disadvantage in performance.

3.2 Performance of Remote I/O Operations

In remote I/O operations, optimization of inter-machine data transfer is a key to improve performance. Performance of inter-machine data transfer using TCP/IP can be optimized by TCP window size [9]. In addition, `TCP_NODELAY` option in `setsockopt` is effective. For this optimization, these parameters can be tuned in a Stampi start-up command. For example, the following command:

```
jmpirun -np 1 -sockbufsize 512 -tcpondelay program
```

initiates a process (program) on a local machine with TCP window size of 512 KByte and `TCP_NODELAY` option in inter-machine data transfer. We measured performance of point-to-point inter-machine data transfer with several TCP window sizes. Through this test, we chose an appropriate TCP window size as being 512 KByte.

In remote I/O test, transfer speed of remote I/O operations from the SR to the VPP was measured. In this test, user processes were initiated on the SR and one or two MPI-I/O processes and a single router process were created on the VPP. The I/O operations were carried out on the VPP using a vendor supplied MPI-I/O library. Table 2 shows performance results for this test. In this table, **np** and **io-*np*** denote the numbers of user processes and MPI-I/O processes, respectively. In most cases, performance for two MPI-I/O processes was greater than that for a single MPI-I/O process. We considered that the performance was improved by using two MPI-I/O processes in local I/O operations as shown in Table 1. But improvement was less than that on the local I/O operation. It was due to inter-machine data transfer time that dominates the remote I/O operations. Thus inter-machine data transfer was a bottleneck in this test.

Inter-machine data transfer in remote I/O operations is realized with a flexible communication library used in the Stampi library. To evaluate those results, we measured performance of inter-machine data transfer using `MPI_Send/Recv` in Stampi. In this test, we used a master-slave type program, where master and slave processes run on the SR and the VPP, respectively. Performance results in

Table 1. Performance results of `MPI_File_write_at_all` and `MPI_File_read_at_all` on a Fujitsu VPP5000

| | np | Transfer rate (MB/s) | | | | | | |
|------------------------------------|----|----------------------|--------|--------|--------|-------|-------|-------|
| | | Data size (Byte) | | | | | | |
| | | 512 K | 1 M | 4 M | 8 M | 16 M | 64 M | 256 M |
| <code>MPI_File_write_at_all</code> | 1 | 300.3 | 305.3 | 305.2 | 1061.0 | 93.6 | 53.5 | 32.9 |
| | 2 | 2155.2 | 2267.6 | 2359.9 | 2373.9 | 136.8 | 106.2 | 66.0 |
| | 4 | 3876.0 | 4273.5 | 4640.4 | 7719.8 | 380.6 | 222.7 | 111.3 |
| <code>MPI_File_read_at_all</code> | 1 | 114.7 | 109.8 | 106.1 | 1272.9 | 103.2 | 81.2 | 55.5 |
| | 2 | 2304.1 | 2427.2 | 2523.7 | 2535.7 | 109.7 | 108.8 | 108.9 |
| | 4 | 4166.7 | 4524.9 | 4956.6 | 5044.1 | 497.9 | 507.6 | 218.7 |

Table 2. Performance results of `MPI_File_write_at_all` and `MPI_File_read_at_all` in remote I/O operations from a Hitachi SR8000 to a Fujitsu VPP5000

| | np | io-np | Transfer rate (MB/s) | | | | | |
|-------------------------------------|----|-------|----------------------|------|------|------|------|-------|
| | | | Data size (Byte) | | | | | |
| | | | 512 K | 1 M | 8 M | 16 M | 64 M | 256 M |
| MPI_File_write_at_all (SR → VPP) | 2 | 1 | 7.67 | 10.2 | 8.63 | 8.72 | 8.23 | 5.82 |
| | 2 | 2 | 7.69 | 10.5 | 13.5 | 14.1 | 13.5 | 9.27 |
| | 4 | 1 | 6.38 | 9.48 | 5.38 | 7.71 | 5.78 | 5.11 |
| | 4 | 2 | 6.82 | 11.2 | 1.60 | 11.5 | 14.8 | 9.12 |
| | 8 | 1 | 1.28 | 1.39 | 0.30 | 1.63 | 1.20 | 2.86 |
| | 8 | 2 | 1.35 | 1.32 | 0.53 | 3.10 | 3.39 | 6.53 |
| MPI_File_read_at_all (SR ← VPP) | 2 | 1 | 7.27 | 10.3 | 8.26 | 13.0 | 9.77 | 8.21 |
| | 2 | 2 | 12.2 | 17.2 | 22.5 | 23.3 | 21.4 | 18.7 |
| | 4 | 1 | 3.94 | 7.40 | 17.1 | 18.5 | 17.9 | 17.1 |
| | 4 | 2 | 7.12 | 13.1 | 19.5 | 17.8 | 20.2 | 18.3 |
| | 8 | 1 | 1.89 | 3.66 | 14.1 | 16.8 | 17.5 | 19.8 |
| | 8 | 2 | 3.76 | 7.22 | 18.7 | 22.5 | 20.8 | 21.9 |

Table 3. Performance results of inter-machine data transfer

| | np(SR) | np(VPP) | Transfer rate (MB/s) | |
|------------------------|--------|---------|----------------------|-------|
| | | | Data size (Byte) | |
| | | | 64 M | 256 M |
| MPI_Send (SR → VPP) | 1 | 1 | 3.63 | 8.40 |
| | 2 | 1 | 2.54 | 6.65 |
| | 4 | 1 | 1.73 | 4.55 |
| | 8 | 1 | 0.908 | 2.35 |
| MPI_Recv (SR ← VPP) | 1 | 1 | 19.7 | 19.1 |
| | 2 | 1 | 22.5 | 22.4 |
| | 4 | 1 | 15.8 | 17.6 |
| | 8 | 1 | 15.9 | 19.5 |

this test are summarized in Table 3, where **np(SR)** and **np(VPP)** are the numbers of processes on the SR and the VPP, respectively. Performance of `MPI_Send` was less than that of `MPI_Recv`. Concerning remote I/O operations, the inter-machine data transfer in write and read operations was realized with `MPI_Send` and `MPI_Recv`, respectively. Therefore, we expected that performance of write operations was less than that of read operations. In addition, performance of `MPI_Send` was poorer with increase of the number of processes on the SR. It seemed that the performance degradation in remote write operations shown in Table 2 was caused by this effect.

4 Summary

We have reported outline, architecture and preliminary performance results of Stampi-I/O. Currently, Stampi-I/O is supported on several platforms, for example, scalar parallel computers (Hitachi SR8000, IBM SP, SGI Origin and Onyx, Cray T3E, etc.), vector parallel computers (Fujitsu VPP5000 and VPP300, NEC SX-4 and SX-5, etc.), WS/PC clusters (Solaris, HP-UX, Linux, FreeBSD, etc.) and so on. In Stampi-I/O, thirty nine MPI-I/O functions are supported.

Through performance measurement tests, we observed efficiency of multiple MPI-I/O processes. On the other hand, it was found that necessity of optimization in inter-machine data transfer was significant for performance improvement in remote write operations. We would like to investigate reasons for the performance degradation by checking TCP/IP communication mechanism in future works.

Finally, the authors would like to thank Dr. G. E. Fagg, Department of Computer Science, University of Tennessee and High Performance Computing Center Stuttgart, for his valuable comments.

References

- [1] Message Passing Interface Forum: “MPI: A Message-Passing Interface Standard”, June 1995. 288
- [2] Message Passing Interface Forum: “MPI-2: Extensions to the Message-Passing Interface Standard”, July 1997. 288
- [3] T. Imamura, Y. Tsujita, H. Koide and H. Takemiya: “An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers”, LNCS 1908, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, pp. 200–207. 288
- [4] R. Thakur, W. Gropp and E. Lusk: “On Implementing MPI-IO Portably and with High Performance”, In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, May 1999, pp. 23–32. 289
- [5] G. E. Fagg, E. Gabriel, M. Resch and J. J. Dongarra: “Parallel IO Support for Meta-computing Applications: MPI-Connect IO Applied to PACX-MPI”, LNCS 2131, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, pp. 135–147. 289
- [6] W. Gropp and E. Lusk: “User’s Guide for mpich, a Portable Implementation of MPI”, (1998), <http://www-unix.mcs.anl.gov/mpi/mpich/>. 289
- [7] R. Thakur, W. Gropp and E. Lusk: “An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces”, In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180–187. 289
- [8] Fujitsu UXP/V MPI User’s Manual V20. 292
- [9] B. Tierney: “TCP Tuning Guide for Distributed Application on Wide Area Networks, *Usenix; login*, February. 2001. 293

(Quasi-) Thread-Safe PVM and (Quasi-) Thread-Safe MPI without Active Polling^{*}

Tomas Plachetka

University of Paderborn, Department of Computer Science
Fürstenallee 11, D-33095 Paderborn, Germany

Abstract. PVM (the current version 3.4) as well as many current MPI implementations force application programmers to use active polling (also known as busy waiting) in larger parallel programs. This serious problem is related to thread-unsafety of these communication libraries. While the MPI specification is very careful in this respect, the implementations are not. We present a new mechanism of interruptable blocking receive which makes PVM and MPI quasi-thread-safe. This mechanism does not require any modifications to the existing semantics of PVM or MPI (we only extend the interfaces with two new functions) and allows writing multi-threaded programs without active polling. Then we sketch how the interrupt mechanism can be hidden in the implementations of PVM and MPI, making both PVM and MPI completely thread-safe without active polling. Results of our experiments promise a significant speedup for all larger communication-intensive parallel applications.

Keywords: PVM, MPI, thread-safety, polling, latency, efficiency

1 Introduction

We define a *non-trivial parallel application* as an application consisting of parallel processes which all perform two independent tasks shown in Fig. 1. Distributed databases, media servers, shared memory simulation libraries and parallel scientific computations using application-independent load balancing libraries are a few examples of important non-trivial parallel applications.

T_1 CPU intensive computation, communicating with other processes

T_2 Fast servicing of requests coming from other processes (in order to provide the other processes with data, or to balance the load, or to handle fatal errors, ...)

Fig. 1. Two independent activities of one process of a non-trivial parallel application

^{*} Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the grant VEGA 1/7155/20

Non-trivial parallel applications based on the current implementations of PVM [4] (PVM 3.4.4) or MPI [3] (e.g. MPICH 1.2.3, ScaMPI 1.13.7) are forced to use *active polling*. Active polling means that the heavy computation in T_1 gets interrupted “now and then” to check for incoming requests in T_2 . If there is no incoming request, then the computation in T_1 is resumed; if there are some requests, they are serviced, and then T_1 is resumed. Here is why it is bad: if “now and then” means often (say, every 1 microsecond) then CPU cycles are unnecessarily wasted every 1 microsecond when there are no incoming requests to service; if “now and then” means seldom (say, every 1 minute) then incoming requests do not get serviced for a minute in the worst case. An obvious compromise is to use a “more reasonable” time period than 1 microsecond or 1 minute (in order to achieve a balance between the wasted CPU time and the message-passing latency). However, the “reasonable” constant must be tuned for every application and for every system on which the application is supposed to run. (Moreover, even a tuned application can exhibit a nondeterministic behaviour.)

Our goal is to avoid active polling in non-trivial applications, not only inside communication libraries. We propose an interrupt-driven mechanism which allows writing non-trivial applications without active polling. This mechanism can either be implemented as an extension to the PVM and MPI interfaces (we refer to leaving the thread synchronisation to the user as to *quasi-thread-safety*) or it can be completely hidden in the PVM and MPI implementations, yielding the highest level of thread-safety defined by the MPI standard (*complete thread-safety*, see `MPI_THREAD_MULTIPLE` in [3])—multiple threads may call PVM/MPI functions, with no restrictions.

We focus on PVM with socket-based communication in this paper. We are aware of different implementations of PVM and MPI and also of different implementations of the MPI specification itself—however, where a reasoning applies to both PVM and MPI (meaning the implementations of the standards), we write “PVM/MPI” throughout the text. We are convinced that the techniques explained in Section 4 can be applied to any implementation of PVM and any implementation of MPI (no matter whether sockets or shared memory are used for inter-process communication).

Section 2 compares our work to previous efforts. Section 3 discusses options of implementing non-trivial applications using PVM. In Section 4 we explain why non-trivial parallel applications cannot be written without active polling using the current PVM/MPI implementations. We present a new interrupt mechanism which makes PVM/MPI quasi-thread-safe and allows avoiding active polling in non-trivial applications. This mechanism is an extension to the PVM/MPI interfaces (two new functions are added to the existing interfaces). The original functionality and performance of PVM/MPI are fully preserved. In the same section we also sketch how the interrupt mechanism can be hidden inside PVM/MPI, making the libraries completely thread-safe, without active polling. Our theoretical considerations are supported by experiments in Section 5.

2 Related Work

Thread-safety of PVM and MPI has been studied for many years. We mention only a few research papers here, the complete list would be too extensive.

LPVM (Lightweight-process PVM) system was introduced in [7]. LPVM is designed for shared-memory machines. PVM tasks are implemented as threads in LPVM. A definition of thread-safety is given in the paper: “A program is thread-safe when multiple threads in a process can be running that program successfully without data corruption. A library is thread-safe when multiple threads can be running a routine in that library without data corruption.” The authors recognize two main issues that have to be dealt with to make PVM multi-thread-safe: *global state and reentrancy*. LPVM removes global states from the PVM library by assigning receive and send buffers (and other resources) to each task. The user interface of PVM (3.3) is only slightly modified but a major redesign of `libpvm` is needed. Our implementation is simple and does not require a removal of global states.

TPVM [2] is a different approach which assumes a thread to be the basic unit of parallelism in a distributed system. There is a thread server registering all threads running in the system. This fine-grained model is mapped onto the coarse-grained process model of PVM for the purpose of message passing. Rather than going into technical details, we shall explain the problem of TPVM from the point of view of a non-trivial application (see Section 4.1 in [2] for more details). There is a global message queue accessible to all threads in each process. A thread wanting to receive a message follows the following protocol. First it looks for the message in the global message queue. If the message is there, the thread continues; if not, the thread attempts to receive a message from another TPVM task using a nonblocking receive. If a message is there, but it is addressed to another thread, the thread stores the message in the global message queue and retries the nonblocking receive (and later wakes up threads waiting for those messages); otherwise it falls asleep. Here is the weakness of the protocol: When a thread falls asleep, then **another** thread must attempt to receive a message in order to wake up the sleeping thread. If there is no such thread, *the sleeping thread will sleep forever*—even though there might be a message addressed to the sleeping thread sent by some other TPVM process. This situation can be resolved by running a special thread that regularly polls for incoming messages and wakes other threads up. Our mechanism does not require running a polling thread.

A PVM library supporting the concept of active messages is described in [6]. The library uses a signal handling mechanism in order to detect a change on the set of receiving socket file descriptors. This idea is close to what the MPICH’s `ch_p4` driver does. However, it does not solve the problem of non-trivial applications.

MiMPI [1], IBM’s MPI and ScaMPI claim to be fully thread-safe. It is difficult to judge without further information how the problem of thread-safety is solved in the implementations. Active polling is used inside ScaMPI.

3 Programming Non-trivial Applications with PVM

This section discusses different programming techniques for implementing non-trivial applications using PVM 3.4 and explains why active polling cannot be avoided.

3.1 PVM and Thread-Safety

Threads are a natural way of implementing the tasks T_1 and T_2 from Fig. 1 to run simultaneously (concurrently) in the scope of a single PVM task (a single process). Fig. 2 a shows a pseudo-code of a generic non-trivial parallel application. We restrict the communication of thread T_1 to sending messages but this restriction is not important. We implemented the pseudo-code of Fig. 2 a. It does not work. The messages sent by `pvm_send` of the thread T_1 get never delivered. PVM does not allow a thread to send a message while another thread is blocked in `pvm_recv`. This is a consequence of “*PVM is not thread-safe*”.

Fig. 2 b shows how this problem can be overcome. Thread T_2 uses active polling to check for incoming messages. Thread T_1 is allowed to call `pvm_send` only during inactive periods of T_2 (when T_2 is blocked in the `sleep` call). This code works; however, it is very poor for the reasons explained in Section 1. Unfortunately, as we show in the following, the pseudo-code of Fig. 2 b is the only way to implement the desired functionality.

3.2 Handlers which Do Not Get Fired Unless They Are Told to Do so

Message handlers have been introduced in PVM 3.4 (see also [5]). They *seem* to provide an alternative way of implementing the scenario of Fig. 2. We briefly explain what message handlers are and what is wrong with them.

| | | | |
|--------------------------------|--------------------------------|--------------------------------|---------------------------------|
| | | mutex comm; | |
| | | | <u>Thread T_2</u> |
| <u>Thread T_1</u> | <u>Thread T_2</u> | <u>Thread T_1</u> | while (not_done) |
| while (not_done) | while (not_done) | while (not_done) | lock(comm); |
| compute(); | <code>recv()</code> ; | compute(); | arrived= <code>probe()</code> ; |
| <code>send()</code> ; | handle_message(); | lock(comm); | while (arrived) |
| | | <code>send()</code> ; | <code>recv()</code> ; |
| | | unlock(comm); | handle_message(); |
| | | | arrived= <code>probe()</code> ; |
| | | | unlock(comm); |
| | | | sleep(time); |

a) Natural implementation, not working b) Active polling, working but poor

Fig. 2. Threaded implementation of one process of a generic non-trivial application

PVM 3.4 users are encouraged to create own message handlers using the function

```
pvm_addmhf(int src, int tag, int ctx, int (*func)(int mid))
```

which registers a user's message handler `func`. The handler `func` is fired when a matching message arrives (the message header must match the parameters `src`, `tag` and `ctx`).

An elegant implementation of the task T_2 from Fig. 1 would be registering a set of message handlers for T_2 , getting even rid of threads (the only running thread would be T_1). However, the manual to `pvm_addmhf` says (note the marked text): "...`pvm_addmhf` specifies a function that will be called *whenever libpvm copies in a message* whose header fields of `src`, `tag`, and `ctx` match those provided to `pvm_addmhf()`". In other words, *the message handlers are called only when the application asks for it* (by calling `pvm_recv` or `pvm_probe`, or `pvm_send`, ...) If the task T_2 is implemented using message handlers registered by `pvm_addmhf`, then T_2 will not receive any message until T_1 wants to communicate!

Using message handlers for implementing non-trivial applications therefore requires running a thread that calls `pvm_probe` (or some other PVM function) at regular time intervals. This equals to active polling.

3.3 Lazy Signaling

PVM allows delivering signals between tasks. Signals, in combination with message handlers, can be used to get rid of the polling thread mentioned in Section 3.2. The scenario of Fig. 1 would work as follows (we present a simplified version here, the actual implementation can be very complex):

1. Process A sends a message to process B
2. Meanwhile, process B runs T_1 , not noticing the message
3. Process A sends a signal to process B , saying "Get up, you got a message!"
4. Process B gets the signal and fires a signal handler. The signal handler calls `pvm_probe` which fires a message handler that receives the message and performs an appropriate T_2 's action.

Note that there is no active polling in the above scenario. However, technical issues make this approach *inefficient and non-portable* and restrict the use of PVM to *homogeneous parallel machines*.

4 (Quasi-) Thread Safe PVM/MPI without Active Polling

The reasons why the scenario from Fig. 1 cannot be efficiently implemented using the current standards are:

1. Tasks T_1 and T_2 must be implemented as threads.
2. T_2 must run a blocking receive.

3. T_1 cannot send any messages while T_2 is blocked in the blocking receive if the communication library (PVM or MPI) is not thread-safe.
4. It seems to be difficult from the software engineering point of view to implement the PVM and MPI libraries in a reentrant way without active polling.

In the following section we first describe an *interrupt mechanism* which is missing in PVM and in MPI and which *allows* to avoid active polling in non-trivial multi-threaded parallel applications—this is what we call *quasi-thread-safety*. Quasi-thread-safety leaves the correct synchronisation of threads to the user (or to a library built on top of quasi-thread-safe PVM/MPI). Then we sketch *how to make PVM/MPI completely thread-safe*, hiding the interrupt mechanism in the implementations of the standards.

4.1 Interruptable Blocking Receive and Quasi-Thread-Safety

Let us return to one of the reasons of PVM/MPI being thread-unsafe unless active polling is used (point 3): “ T_1 cannot send any messages while T_2 is blocked in the blocking `recv` if the communication library (PVM or MPI) is not thread-safe.” In terms of concurrent programming, T_2 is waiting inside a critical section of a blocking `recv` which does not allow T_1 to enter *its* critical section of a `send`. Our idea is to let T_1 *interrupt* T_2 for the time necessary for completing the `send` call. T_2 must be blocked outside of its critical section during the interrupt. Care must be taken to not let T_1 enter its critical section before T_2 has left its critical section. At the end of the interrupt (after the `send` call has returned), after T_1 has left its critical section, T_2 must return to exactly the same state in which it was before the interrupt.

To implement the above mechanism, we extended the PVM (3.4) and MPI (MPICH 1.2.3, `ch_p4`) libraries with two new functions. The extensions require only a few changes in the implementation of PVM/MPI (literally, a few lines of code), while the original functionality does not change at all (in case of PVM only the implementation of the library is slightly modified, the PVM daemon is not changed). This is the C interface and semantics of the new functions:

```
void interrupt_recv(void)
```

Blocks the calling thread until the thread that is blocked in a `recv` has blocked outside of its critical section and then returns the control to the calling thread. (If there is no thread blocked in a `recv`, the control never returns to the calling thread.)

```
void resume_recv(void)
```

Makes the thread blocked outside of the critical `recv`’s section reenter its critical `recv`’s section and then returns the control to the calling thread.

The extended PVM/MPI remain generally thread-unsafe. A random sequence of concurrent calls to PVM/MPI functions results in an undefined behaviour. We do not intend to call the functions randomly. We restrict ourselves

to calling PVM/MPI functions in a well defined order that avoids both thread-safety problems and active polling when writing non-trivial parallel applications. This is possible to achieve with the extended PVM/MPI and this is why we call the extended libraries quasi-thread-safe. To simplify writing of programs that use only the well defined order of PVM/MPI calls, we developed a programming paradigm implemented as a library (TPL, Thread Parallel Library) that uses the strategy described at the end of this section.

Fig. 3 depicts a modified scenario of Fig. 2 a, revealing the integration of `interrupt_recv` and `resume_recv` in PVM/MPI in more detail. This scenario avoids active polling as well as thread-safety problems in PVM/MPI. The trick is based on sending a fake “message” from a process to itself. We assume a socket inter-process communication here (but the same interrupt mechanism can be implemented for shared memory communication as well). The implementation of `recv` in PVM/MPI causes T_2 to block in a `select` call on a set of file descriptors connected to sockets (leading to other processes in the network). We extend this set of file descriptors with the read-end of a synchronous POSIX pipe `intr_fd`. The function `interrupt_recv` called by T_1 simply writes to the write-end of `intr_fd`. This `write` causes T_1 to block until the read-end of the (synchronous) pipe has been read. The read-end of `intr_fd` becomes readable, the `select` in `recv` unblocks and T_2 bails out of its critical section. After that, `recv` reads from `intr_fd` and T_2 gets blocked in the following `read`. Now the thread T_1 gains control and can safely send a message. After the message has been sent, T_1 calls `resume_recv` which is implemented in Fig. 3 in exactly the same way as `interrupt_recv`. `resume_recv` writes to the write-end of `intr_fd`. This unblocks the second `read` in `recv` and T_2 gets eventually blocked in the `select` of its critical section again. (The implementation of “bailing out of the critical `recv` section” is tricky. At the time of writing we have a solution for socket-based PVM 3.4 and MPICH’s `ch_p4` driver.)

Here is a sketch of how PVM/MPI can be made completely thread-safe, without a loss of efficiency caused by active polling:

- Each PVM/MPI process runs a thread (let us call it *main thread*) that is (almost always) blocked in a blocking `recv`, listening to all sockets from where a message might arrive. When a message arrives, it is stored by main thread in a global message queue (or message queues).
- The message queue is protected by a mutex.
- Each `send` is preceded by `interrupt_recv` and followed by `resume_recv`.
- All `send` calls are mutually excluded (using a mutex).
- The user’s code runs as a thread (or as several threads if the application itself is multi-threaded). User’s `recv` calls are implemented as functions that access the message queue, not file descriptors. A `recv` gets blocked on a semaphore when it did not find the requested message.
- All PVM/MPI functions are divided into collision classes. Each class contains functions that internally modify the same global memory structures (or cause some other racing conditions when called simultaneously). Functions of one class are mutually excluded (using a mutex).

- Thread addressing should not be a part of PVM/MPI interface. It is up to the user to develop a thread addressing scheme (if it is needed in the application).

5 Experiments

We used two versions of the pingpong benchmark in our experiments. The pingpong application consists of two processes, P_1 and P_2 . In both versions the process P_2 runs a loop in which it first receives a message and then answers it. The two threaded versions of the process P_1 are depicted in Fig. 2 b (*POLL*, the active polling version, based on thread-unsafe PVM/MPI) and Fig. 3 (*QTS*, the event-driven version based on quasi-thread-safe PVM/MPI). In both versions there is no `compute()` call in T_1 and no `handle_message()` call in T_2 . We used a `nanosleep` of 50 milliseconds in the active polling version (as we show later, the choice of this constant is optimal—a finer setting has the same effect).

Table 1 shows the time measurements in which the process P_1 sent (and received) 100,000 messages of the length 10,000 Bytes. Each measurement was repeated 10 times. “Maximum deviation” is the absolute maximum deviation from the average time (in seconds). We also observed the number of `nanosleep()` calls which explains the timing differences between the polling versions of the

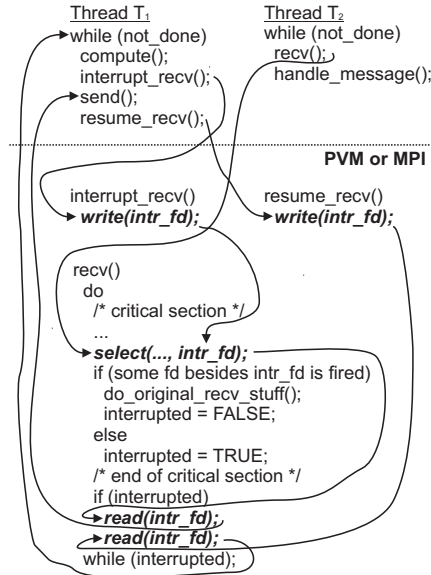


Fig. 3. Scenario of Fig. 2 a using the interrupt mechanism of quasi-thread-safe PVM/MPI. The synchronisation points (where a thread can possibly block) are typeset in *slanted boldface*. A synchronous pipe `intr_fd` is used here for the thread synchronisation

Table 1. Pingpong benchmark, 100,000 messages

| Communication library Pingpong version | PVM POLL | PVM QTS | MPICH POLL | MPICH QTS | ScaMPI POLL | ScaMPI TS |
|---|-------------|------------|---------------|--------------|----------------|--------------|
| Average time (in seconds) | 120 | 21 | 42 | 26 | 98 | 339 |
| Maximum deviation (in seconds) | ± 66 | ± 3 | ± 1 | ± 1 | ± 40 | ± 81 |
| Number of <code>sleep()</code> calls | 2,370 | 0 | 583 | 0 | 1,533 | 0 (?) |

benchmark (see below). The data were measured on a double-processor 850 MHz Intel Pentium 3 running Linux Redhat. We compared PVM 3.4, MPICH 1.2.3 (ch_p4 driver) and ScaMPI 1.13.7. ScaMPI is thread-safe, therefore we could also use the code from Fig. 2 a (*TS*) for a direct comparison with the quasi-thread-safe MPICH's version. All times are absolute execution times (the timing starts right before the receiving loop in the process P_1 and stops right after the loop).

Note that the active polling versions of the pingpong benchmark are *profiting* from not calling `sleep` (`nanosleep`) very often. A continuous stream of messages sent by the process P_2 allows the process P_1 to pull the messages from the network one after another without falling asleep after each `recv`. The deviations in the measured times by the POLL versions are caused by the varying number of `sleep()` calls in the individual runs of the benchmark. If the stream of messages is not continuous, then the latency of each `recv` followed by a `sleep` depends *solely* of the timer's resolution and the process/thread switching overhead in the operating system (not of the speed of the processors or the network connecting them). Without a special tuning of the kernel the latency is 0.02 sec (the `time` argument of the `nanosleep(time)` call is not important). This can be measured by calling `nanosleep(1 microsecond)` many times in a loop.

6 Conclusions

We showed that the current PVM implementation and some of MPI implementations force a whole class of important parallel applications to use active polling. We also showed the limitations of active polling in non-trivial communication-intensive applications. We explained a new interrupt mechanism that makes PVM as well as MPI quasi-thread-safe (quasi-thread-safety means a possibility of writing non-trivial applications without active polling) or, with a little more effort, completely thread-safe. No change in the interfaces of PVM or MPI is needed in order to achieve complete thread-safety of both standards without active polling. We demonstrated the potential of our interrupt mechanism on a pingpong benchmark. The benchmark is an abstraction of all non-trivial parallel applications. Even in a comparison to a highly optimised active polling we measured significant speedups when we used the new interrupt mechanism of quasi-thread-safe PVM 3.4 and quasi-thread-safe MPICH 1.2.3. Moreover, we measured a speedup of 13.5 in a direct comparison to the commercial thread-safe MPI implementation by Scali.

Acknowledgements

We would like to thank Burkhard Monien and all the people in his working group, among them especially to Axel Keller for his perfect technical support and Ulf-Peter Schroeder for his valuable hints. Our thanks go also to Branislav Rován and Peter Ružička at the Comenius University in Bratislava who many years ago turned our attention to parallel and distributed programming.

References

- [1] F. García, A. Calderón, and J. Carretero. MiMPI: A multithread-safe implementation of MPI. In *Proceedings of PVM/MPI 99, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes on Computer Science*, pages 207–214. Springer Verlag, 1999. 298
- [2] A. Ferrari and V.S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. *Concurrency—Practice and Experience*, 10(3):199–228, 1998. 298
- [3] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. 297
- [4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine (A User's Guide and Tutorial for Networked Parallel Computing)*. The MIT Press, 1994. 297
- [5] A. Geist, J. A. Kohl, P. M. Papadopoulos, and S. Scott. Beyond PVM 3.4: What we've learned, what's new and why. In *Proceedings of PVM/MPI 97, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes on Computer Science*, pages 116–126. Springer Verlag, 1997. 299
- [6] K. R. Subramaniam, S. C. Kothari, and D. E. Heller. A communication library using active messages to improve performance of PVM. *Journal of Parallel and Distributed Computing*, 39(2):146–152, 1996. 298
- [7] H. Zhou and A. Geist. LPVM: A step towards multithread PVM. *Concurrency—Practice and Experience*, 10(5):407–416, 1998. 298

An Implementation of MPI-IO on Expand: A Parallel File System Based on NFS Servers

Alejandro Calderón, Félix García, Jesús Carretero,
Jose M. Pérez, and Javier Fernández

Computer Architecture Group, Computer Science Department
Universidad Carlos III de Madrid
Leganés, Madrid, Spain
`fgarcia@arcos.inf.uc3m.es`

Abstract. This paper describes an implementation of MPI-IO using a new parallel file system, called Expand (Expandable Parallel File System)¹, that is based on NFS servers. Expand combines multiple NFS servers to create a distributed partition where files are declustered. Expand requires no changes to the NFS server and uses RPC operations to provide parallel access to the same file. Expand is also independent of the clients, because all operations are implemented using RPC and NFS protocol. The paper describes the design, the implementation and the evaluation of Expand with MPI-IO. This evaluation has been made in Linux clusters and compares Expand and PVFS.

Keywords: Parallel File System, NFS, data declustering, clusters, `b_eff_io`.

1 Introduction

This paper shows a new approach to the building of parallel file systems for heterogeneous clusters. The result of this approach is a new parallel file system, called Expand (*Expandable Parallel File System*). Expand allows the transparent use of multiple NFS servers as a single file system. Different NFS servers are combined to create a distributed partition where files are declustered. Expand requires no changes to the NFS server and it uses RPC operations to provide parallel access to the same file. Using this system, we can join different NFS servers (Linux, Solaris, Windows 2000, etc.) to provide parallel access to files in a heterogeneous cluster.

The rest of the paper is organized as follows: Section 2 presents the Expand design, where we describe the data distribution, the structure of the files, the naming and metadata management and the parallel access to files. Performance evaluation is presented in Section 4. The evaluation has been made using a Linux cluster and compares Expand with PVFS. Finally, Section 5 summarizes our conclusions and the future work.

¹ This work has been partially support by the Spanish Ministry of Science and Technology under the TIC2000-0469 and TIC2000-0471 contracts, and by the Community of Madrid under the 07T/0013/2001 contract.

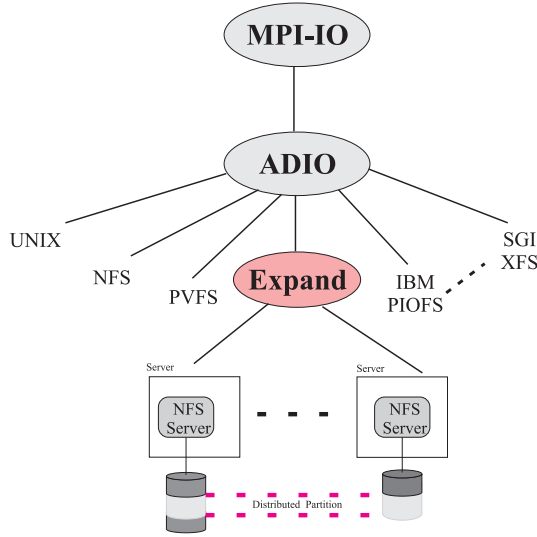


Fig. 1. Expand Architecture

2 Expand Desing

The main motivation of the Expand design is to build a parallel file system for heterogeneous clusters. To satisfy this goal, authors are designing and implementing a parallel file system using NFS servers, whose first prototype is described in this paper.

Network File system (NFS) [16] supports the NFS protocol, a set of remote procedure call (RPC) that provides the means for clients to perform operations on a remote file server. This protocol is operating system independent. Developed originally for being used in networks of UNIX systems, it is widely available today in many systems, as Linux or Windows 2000, two operating systems very frequently used in clusters.

Figure 1 shows the architecture of Expand, and the integration inside ROMIO. This figure shows how multiple NFS servers can be used as a single file system. File data are declustered by Expand among all NFS servers, using blocks of different sizes as stripping unit. Processes in clients use an Expand library to access to an Expand distributed partition. Expand offers an interface based of POSIX system call. This interface, however, is not appropriate for parallel applications using strided patterns with small access size [11]. For parallel applications, we are integrating Expand inside ROMIO [14] to support MPI-IO interface, implementing the appropriate Expand ADIO. Figure 1 illustrates this integration.

Using the former approach offers the following advantages:

1. No changes to the NFS server are required to run Expand. All aspects of Expand operations are implemented on the clients.

2. Expand is independent of the operating system used in the client. All operations are implemented using RPC and NFS protocol.
3. The parallel file system construction is greatly simplified, because all operations are implemented on the clients. This approach is completely different to that used in many current parallel file system as CFS [13], Vesta [3], HFS [8], PIOUS [10], Scotch [5], PPFS [9], ParFiSys [2], Galley [11], and PVFS [1].
4. It allows the use of servers with different architectures and operating systems, because the NFS protocol hides those differences.
5. Simplifies the configuration, because NFS is very familiar to users. Server only need to export the appropriate directories and clients only need a little configuration file that explain the distributed partition.

There are other systems that use NFS servers as basis of their work, similarly to Expand. Bigfoot-NFS [7], for example, also combines multiples NFS servers. However, this system uses files as the unit of interleaving (i.e., all data of a file reside in one server). Although files in a directory might be interleaved across several machines, it does not allow parallel access to the same file. Other similar system is Slice file system [4]. Slice is a storage system for high-speed networks that uses a packet filter μ proxy to virtualize the NFS, presenting to NFS clients a unified shared file volume. This system uses the μ proxy to distribute file service requests across a server ensemble and it offers compatibility with file system clients. However, the μ proxy can be a bottleneck that can affect the global scalability of the system.

Next sections describe data distribution, file structure, naming, metadata management, and parallel access to files in Expand.

2.1 Data Distribution and Files

Expand combines several NFS servers (see figure 2) to provide a generic striped partition. Each server exports one or more directories that are combined to build a distributed partition. All files in the system are declustered across all NFS servers to facilitate parallel access, with each server storing conceptually a subfile of the parallel file.

A file in Expand consists in several *subfiles*, one for each NFS partition. All subfiles are fully transparent to the Expand users. On a distributed partition, the user can create, in the current prototype, *striped files with cyclic layout*. In these files, blocks are distributed across the partition following a round-robin pattern. This structure is shown in Figure 2.

2.2 Naming and Metadata Management

Each subfile of a Expand file (see Figure 2) has a small header at the beginning of the subfile. This header stores the file's metadata. This metadata includes the following information: *stride size*, *base node* (identifies the NFS server where the first block of the file resides) and *Round-robin pattern*.

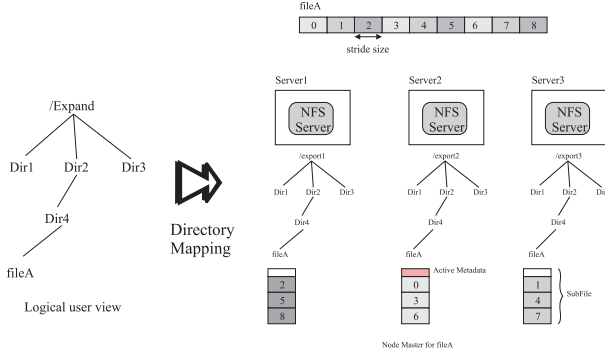


Fig. 2. Directory mapping and Expand file with cyclic layout

All subfiles have a header for metadata, although only one node, called *master node* (described below) stores the current metadata. The master node can be different from the base node. To simplify the naming process and reduce potential bottlenecks, Expand does not use any metadata manager, as the used in PVFS [1]. Figure 2 shows how directory mapping is made in Expand. The Expand tree directory is replicated in all NFS servers. In this way, we can use the lookup operation of NFS without any change to access to all subfiles of a file.

The metadata of a file resides in the header of a subfile stored in a NFS server. This NFS server is the *master node* of the file, similar to the mechanism used in the Vesta Parallel File System [3]. To obtain the master node of a file, the file name is hashed into the number of node: $hash(namefile) \Rightarrow NFSserver_i$.

The use of this simple approach offers a good distribution of masters, balancing the use of all NFS servers and, hence, the I/O load. Initially the base node for a file agrees with the master node. Because the determination of the master node is based on the file name, when a user renames a file, the master node for this file is changed.

2.3 Parallel Access

NFS clients use a *filehandle* to access the files. A NFS filehandle is a opaque reference to a file or directory that is independent of the filename. All NFS operations use a filehandle to identify the file or directory which the operation applies to. Only the server can interpret the data contained within the filehandle. Expand uses a *virtual filehandle*, that is defined as: $\bigcup_{i=1}^N filehandle_i$, where $filehandle_i$ is the filehandle used for the NFS server i to reference the subfile i belonging to the Expand file. The virtual filehandle is the reference used in Expand to reference all operations. When Expand needs to access to a subfile, it uses the appropriated filehandle. Because filehandles are opaque to clients, Expand can use different NFS implementations for the same distributed partition.

To enhance I/O, user requests are split by the Expand library into parallel subrequests sent to the involved NFS servers. When a request involves k NFS

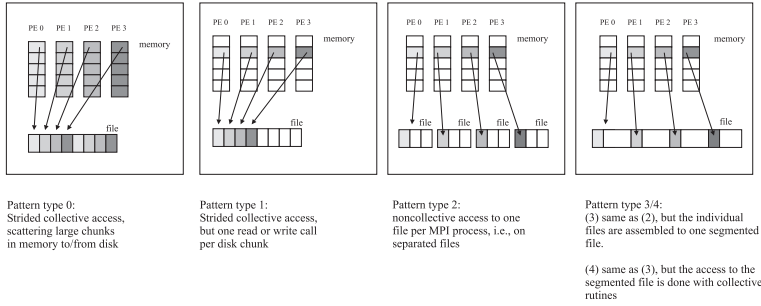


Fig. 3. Data transfer patterns used in `b_eff_io`. Each diagram shows the data transferred by one MPI-IO write call

servers, Expand issues k requests in parallel to the NFS servers, using threads to parallelize the operations. The same criteria is used in all Expand operations. A parallel operation to k servers is divided in k individual operations that use RPC and the NFS protocol to access the corresponding subfile.

3 Evaluation

To evaluate Expand with MPI-IO we have use two different workloads:

- Strided accesses.
- Effective File-I/O bandwidth benchmark (`b_eff_io`).

In the strided access tests, multiple noncontiguous regions of the data file were simultaneously accessed by each process. We used a file of 100 MB and different access size (from 512 bytes to 64 KB).

The effective File-I/O bandwidth benchmark (`b_eff_io`) [15] covers two goals: to achieve a characteristic average number for the I/O bandwidth achievable with a parallel MPI-IO applications, and to get detailed information about several access patterns and buffer lengths. The benchmark examines *first write*, *rewrite* and *read* access, strided (individual and shared pointer) and segmented collective patterns on one file per application and non-collective access to one file per process (see Figure 3).

3.1 Experimental Results

The platform used in the evaluation was a cluster with 8 Pentium III biprocessors, each one with 1GB of main memory, connected through a Fast Ethernet, and executing Linux operating system (kernel 2.4.5). All experiments have been executed on Expand and PVFS [1] using MPI-IO as interface. For both parallel file system, we have used 8 I/O servers. The block size used in all files, both in Expand as PVFS tests, has been 8 KB. All clients have been executed in the 8 machines, i.e: all machines in the cluster have been used as server and clients.

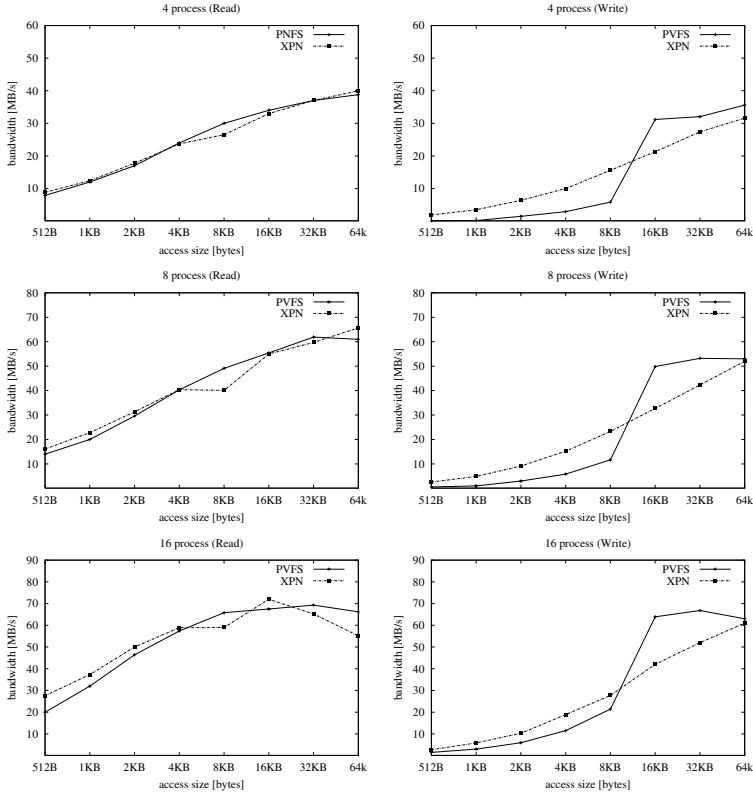


Fig. 4. Results for strided accesses

The results for strided accesses are shown in Figure 4. Figures show the aggregated bandwidth obtained for write operations and read operations for 4, 8, and 16 processes and different access size. For read operations, the performance is very similar in both parallel file system. PVFS offers best results in write operations for large access size. A special detail of the evaluation is the poor performance obtained by PVFS in the cluster used in the evaluation for short access size (lower than 1 KB).

Figure 5 and 6 present the results obtained run `b_eff_io` benchmark in Expand and PVFS. The test have been scheduled to run $T = 10$ minutes. The three diagrams in each row of Figures 5 and 6 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 tupe and as a function of the chunk size. The tests have been done with individual file pointers.

Performance results are satisfactory and demonstrate that the use of our parallel file system with MPI-IO can be a good solution for clusters.

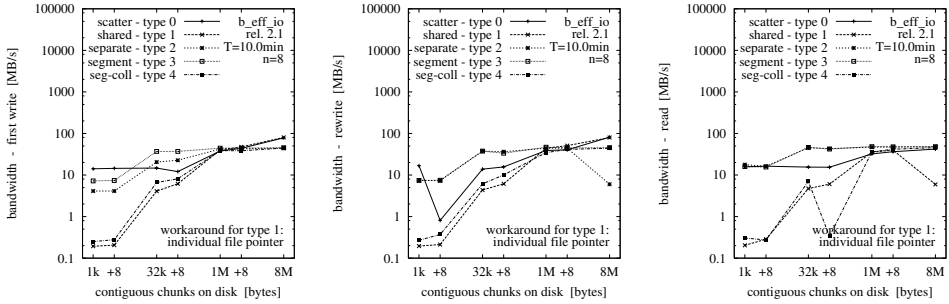


Fig. 5. Results for b_eff.io benchmark. 8 PE in XPN

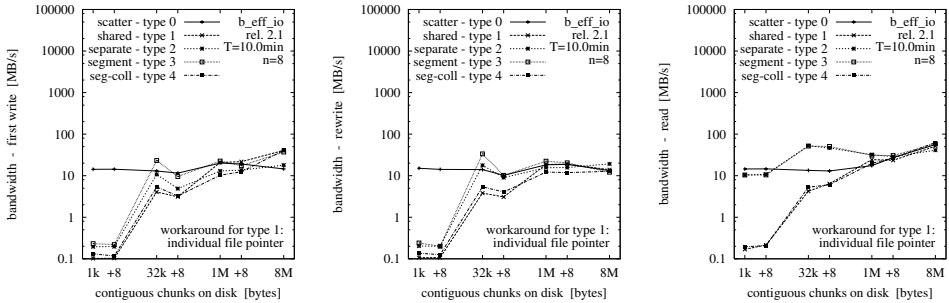


Fig. 6. Results for b_eff.io benchmark. 8 PE in PVFS

4 Conclusions and Future Work

In this paper we have presented the design of a new parallel file system, named Expand, for clusters of workstations. Expand is built using NFS servers as basis. This solution is very flexible because no changes to the NFS servers are required to run Expand. Furthermore, Expand is also independent of the operating system used in the clients, due to the use of RPC and NFS protocol. Expand combines several NFS servers to provide a distributed partition where files are declustered. Expand has been included in ROMIO by implementing a new ADIO. The evaluation has compared the performance of Expand with the obtained in PVFS with different workloads. Further work is going on to optimize Expand and to support fault tolerance.

References

- [1] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Takhur. PVFS: A Parallel File System for Linux Clusters. Technical Report ANL/MCS-P804-0400, 2000. 308, 309, 310

- [2] J. Carretero, F. Perez, P. de Miguel, F. Garcia, and L. Alonso. Improving the Performance of Parallel File Systems. *Parallel Computing: Special Issue on Parallel I/O Systems. Elsevier*, (3):525–542, April 1997. 308
- [3] P. Corbett, S. Johnson, and D. Feitelson. Overview of the Vesta Parallel File System. *ACM Computer Architecture News*, 21(5):7–15, December 1993. 308, 309
- [4] J.S. Chase D.C. Anderson and A.M. Vahdat. Interposed request routing for scalable network storage. In *Fourth Symposium on Operating System Design and Implementation (OSDI2000)*, 2000. 308
- [5] G. Gibson. The Scotch Paralell Storage Systems. Technical Report CMU-CS-95-107, Scholl of Computer Science, Carnegie Mellon University, Pittsburgbh, Pennsylvania, 1995. 308
- [6] J. Huber, C. L. Elford, and et al. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. IEEE, July 1995.
- [7] G. H. Kim and R. G. Minninch. Bigfoot-NFS: A Parallel File-Striping NFS Server. Technical report, Sun Microsystems Computer Corp., 1994. 308
- [8] O. Krieger. *HFS: A Flexible File System for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994. 308
- [9] T. Madhyastha. *Automatic Classification of Input/Output Access Patterns*. PhD thesis, niversidad de Illinois, Urbana-Champaign, 1997. 308
- [10] S.A. Moyer and V.S. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High-Performance Computing Conferece*, pages 71–78, 1994. 308
- [11] N. Nieuwejaar and D. Kotz. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing*, May 1996. 307, 308
- [12] R. Olfield and D. Kotz. The armada parallel file system, 1998. <http://www.cs.dartmouth.edu/~dfk/armada/design.html>.
- [13] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In John L. Gustafson, editor, *Proceedings of the Fourth Conference on Hypercubes Concurrent Computers and Applications*, pages 155–161. HCCA, March 1989. 308
- [14] W. Gropp R. Takhur and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 23–32, 1999. 307
- [15] R. Rabenseifner and A. E. Koniges. Effective Communication and File-I/O Bandwidth Benchmark. In J. Dongarra and Yiannis Cotronis (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, proceedings of the 8th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2001, Santorini, Greece, Sep. 23-26, LNCS 2131, pp 24-35. 310
- [16] R. Sandberg, D. Goldberg, S. Kleiman, D Walsh, and B. Lyon. Design and Implementation of the SUN Network Filesystem. In *Proc. of the 1985 USENIX Conference*. USENIX, 1985. 307

Design of DMPI on DAWNING-3000

Wei Huang, Zhe Wang, and Jie Ma

Institute of Computing Technology, Chinese Academy of Sciences
P.O. Box 2704, Beijing 100080, P.R.China
`{hw,majie,wz,md,snh}@ncic.ac.cn`

Abstract. This paper introduces the design and implementation of DMPI, a high performance MPI running on a cluster of SMPs (CLUMPS) called DAWNING-3000 super-server. DMPI provides high performance communication layer with remote memory access (RMA) functions and a flexible and sufficient extension of ADI (DADI-E). The bandwidth of DMPI is almost reached the peak bandwidth of the lower level communication protocol. With the expanded DADI-E, it is easily to provide a high performance PVM library with the efficient lower communication layer of MPI. Important features of DMPI and DADI-E are presented in the paper, including special data path, RMA mechanism, dynamic process management and special support for PVM. The performance of DMPI over Myrinet is also given.

1 Introduction

MPICH, developed by Argonne National Laboratory, U.S.A, is an important implement of MPI standard. It can easily be ported to other MPI implementation over different low-level communication protocols. The MPICH implementation of the MPI standard is built on a lower level communications layer called ADI (abstract device interface). The ADI provides basic, point-to-point message-passing services for the MPICH implementation of MPI. The MPICH code handles the rest of the MPI standard, including the management of datatypes and communicators, and the implementation of collective operations with point-to-point operations.

DMPI is a high-performance MPI designed and implemented on DAWNING-3000. DAWNING-3000 is a CLUMPS consisting of 70 nodes. Each node is a 4-way 375MHz Power3 SMP, running IBM AIX4.3.3. All nodes are connected with Fast Ethernet, Myrinet. DMPI is based on Resource Management System (RMS) and Basic Communication Library (BCL), which are developed on our own system. A few important features of DMPI are described as follows.

- Special data path. DMPI provides a special data path to improve the communication bandwidth. Since the lower level communication layer can provide high bandwidth and low latency, the normal data path used in MPICH is not efficient enough. DMPI provides an efficient data path to improve the communication performance.

- RMA mechanism. RMA is a new module in MPI-2 standard. It is a complement to the point-to-point message-passing programming model. This new model facilitates the coding of some application with dynamically changing data access patterns where the data distribution is fixed or slowly changing.
- Dynamic process management. Dynamic process management is another new module provided in MPI-2 standard. In this new model, new processes can be dynamically created and management after an MPI application has been started.
- Flexible and sufficient DADI-E. DADI-E is implemented on DAWNING-4000, which extends the ADI-2 to support DPVM and MPI-2 standard. normally, PVM is base on TCPIP communication protocol. In our design, PVM is base on DADI-E, and is named DPVM. So that the DPVM can benefit from the efficient communication data path using DADI-E.

Section 2 gives an overview of the communication software used on our system, including BCL, DPVM and DMPI. In the following sections, DMPI is discussed in detail, including an optimization that can obviously increase the bandwidth of long message in the point-to-point communication, and the implementation of RMA operation and dynamic process management. Then the evaluation of DMPI on DAWNING-3000 platform is shown in Section 4. Finally, we present our conclusions and discuss ongoing research in Section 5.

2 Overview

BCL is a low-level communication software used on DAWNING-3000. The current version of BCL is BCL-3. It is implemented on Myrinet. The upper levels can make full use of the hardware performance via BCL-3. BCL-3 supports point to point message passing. All other collective message passing should be implemented in the higher level software. With zero memory copy and other mechanisms, the performance of BCL-3 can almost reach that raw performance of Myrinet.

RMS is the resource management software used on DAWNING-3000, all of its management work is done by two daemon applications. They are RMD and CSD. RMD is a resource management daemon and CSD is a communication service daemon. RMD manages all the system's resource including nodes, ports, pools, and parallel tasks. RMD takes charge in adding, finishing and killing parallel tasks. It also takes part in creating, modifying and deleting pool. CSD charges for adding tasks. It also provides information to each process running in the parallel task. When a process's information is changed, CSD then broadcast the changes to other processes in the same group.

Fig. 1 shows the protocol stack from BCL-3 applications' point of view. Applications can directly access the BCL-3 level or use other functionality levels. BCL-3 is the lowest level software in DAWNING-3000's communication software. MPI is implemented directly on BCL-3. PVM is implemented on ADI-2

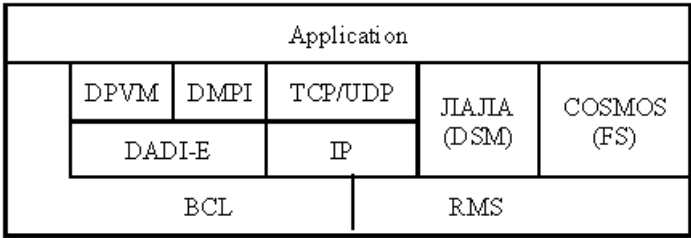


Fig. 1. Protocol stack of DAWNING-3000 software

(Abstract Device Interface) of MPI. Notice that this may be changed according to the different implementations that are done. TCP/IP is designed to port onto our BCL-3 in the next step. A prototype has been completed now. The other two components in the stack are JIAJIA and COSMOS. The former is an DSM software, and the latter is a distributed file system of DAWNING-3000.

Our DMPI is ported from MPICH-1.2.1.10, while DPVM is ported from PVM-3.3.1.1. They are both based on BCL and RMS. Origin PVM used TCP/IP as its communication protocol, for TCP/IP is a general communication protocol and can adapt to a good many of environments both of homogeneous and of heterogeneous. But it is too complex to provide high performance when it is used in some special system such as the DAWNING-3000 super-servers. DMPI and DPVM are implemented to use BCL communication protocol to improve the performance. To benefit from the high performance ADI layer, ADI is extended to DADI-E to support DPVM. DADI-E remains all the functionality of ADI, and adds some new features as follows.

- Add communication calls provided by BCL, which provide the upper communication software MPI and PVM of low-level communication APIs. So that the upper level software can make full use of the hardware performance.
- Add resource management calls provided by RMS. It can be easy for PVM to manage process dynamic resource. And it also is propitious to extend MPI-1 standard to MPI-2 with the new functionality of dynamic process management.
- Provide common APIs both for MPI and PVM. The DADI-E provides basic, point-to-point message-passing services and resource management services for DMPI and DPVM. The users can use MPI or PVM easily, while not knowing how it is really implemented.

RMA and dynamic process management have been implemented in DMPI. All the implementations lie in the DADI-E layer. They will be discussed in detail in the next section.

3 Design Issues

3.1 Performance Optimization

DMPI is a high level communication software. It's driver, BCL, is relatively a low level communication. When implementing DMPI, the high bandwidth and low latency of BCL can't be completely made use of. Figure 2 shows the bandwidth gap between DMPI and BCL. We can see that the peak bandwidth of DMPI is about 180MB/s, which is only 75% to the peak bandwidth of BCL, about 240MB/s.

Because there is no memory copy operation from level of MPI to level of BCL, there should be for some other reasons. When analyzing the procedure flow of DMPI, we find the reason may be that there are too much idle times during each whole data transferring.

The maximum data can be sent by BCL_Send is limited to 128KB, While the size of data message is of no limit when using MPI_Send. If a large data message's size is larger than 128KB, then we should cut the data message into data fragments. Each data fragment is size of 128KB, except that the last data fragment's size maybe is less than 128KB. Then we will send the data fragments by to by. A whole sending/receiving procedure can be described as such:

- Step 1: The sender sends a control message to the receiver, and tells the receiver that he is ready to send a data message.
- Step 2: The receiver has received the control message, and knows that the sender will send to him a data message, so he posts his receiving buffer to the net card. Then he sends back an ack control message packed with the receiving buffer address to the sender.
- Step 3: The sender received the ack control message and got the target address, then he calls the BCL_Send to send the data message to the target buffer.

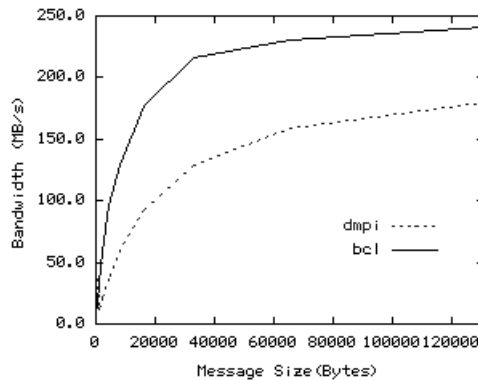


Fig. 2. Bandwidth of DMPI and BCL

- Step 4: The sender does polling the sent complete event and the receiver does polling the received complete event.
- Step 5: When the receiver have polled the received complete event (it means that the data have been received), if all the data fragments have been received at this time, the procedure will be end. Otherwise, goto 2.

The Myrinet net card is able to support for full duplex mode communication. To make full use of the net card, we put forward a thought of pipeline sending. This arithmetic will be explain from behind.

From the procedure described above, if we send a $n \times 128\text{KB}$ large data message, there will be n times handshake between sender and receiver, every data fragment is sent after the preceding one is really received. We assume one handshake will spend x (us) times, and all the other function will spend total y (us) times. So when we send such a large data message, it will spend total $(y + n \times x)$ (us) times. Because when a data fragment is sending, both the sender and receive are idling to wait the sending complete. If we can make use of this idling time to complete their handshake operation, there will need only $(y + x)$ (us) times to complete the total procedure, $(n - 1)$ (us) times less than before. This way is a little like the pipeline in some factory, so we name it pipeline-sending.

Finally, we realized the pipeline sending on DMPI. With this optimization, we improve the bandwidth of large data message as expected. When the data size is 2MB, the bandwidth of DMPI can be improved to 230MB/s, about 95% of BCL's bandwidth.

In the future's work, we will extend this arithmetic to optimize the collective operations, such as `MPI_Bcast`. The current implement of `MPI_Bcast` uses a fairly basic recursive subdivision algorithm. The root sends data to the process size/2 away; the receiver becomes a root for a subtree and applies the same process. When handling the long message, we can do pipelining-sending the messages in data fragments so that the message can be pipelined through the network without waiting for the subtree roots to receive the entire message before forwarding it to the other processors.

3.2 Implementation of RMA

RMA (Remote Memory Access) is one of the important extended components of MPI-2 standard. RMA extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. It is a complement to the point-to-point message-passing programming model, which needs the both sides (source side and target side) to cooperate for completing a communication.

In the communication model of RMA, each process can compute what data it needs to access or update at other processes. However, processes may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes. Thus, the transfer parameters are all available only on one side. This mode of communication facilitates the coding of some applications with dynamically changing data access

patterns where the data distribution is fixed or slowly changing. One another advantage of RMA is that it can save the handshake time in a communication needed by point-to-point message-passing programming model.

At the level of BCL, it has been extended to provide RMA operations. Thus, the DMPI's RMA operations are directly based on the RMA operations of BCL. BCL provides APIs for RMA of DMPI in DADI-E, such as `BCL_Rma_put`, `BCL_Rma_get`, `BCL_Win_create`. Implementation of RMA in DMPI can be partitioned into two parts: Window Management and RMA Communication.

In RMA, all the communication operation is done through an object: "Window", a "window" represents a memory buffer that is allocated only for RMA operations. Window Management consists of such operations as Window Creation, Window Freeing, Setting Window attribute, and so on. The call of `MPI_Win_create` is done to create a new window, and a win id is returned to present for the window. Except at the course of RMA communication, any attribute of a window can be modified. The changed information may be broadcast by the synchronization operations. When an RMA communication is completed, the call of `MPI_Win_free` should be called to free the window. In fact, the object can be shared by many communication calls. So before it is freed, the reference count must be checked to be zero.

RMA communication operation includes three types of communication calls: `MPI_Put`, `MPI_Get`, `MPI_Accumulate` and some different synchronization calls. `MPI_Put` transfers data from the caller memory (origin) to the target memory; `MPI_Get` transfers data from the target memory to the caller memory; and `MPI_Accumulate` updates locations in the target memory. These three calls are all non-blocking, and should cooperate with the synchronization calls, such as `MPI_Fence` and `MPI_Start`.

Because RMA operations are all done by one side, that has got both the source data address and the target address before it is ready to begin communication, so there will be no need to have some handshake for exchange data buffer address between sender and receive as before. Consequently, the implement of RMA communication functions can a little easier than that of point-to-point communication. But there still need some structure, which we call `RMA_FIFO` and `RMA_HANDLE`, to record the information of send request and receive request. BCL provide a common poll call both for point-to-point message passing and RMA to poll the send/receive complete events. When a complete event is polled, it will continue to do next work according to the event type.

In RMA communication operation, another important part is synchronization calls. The function of these calls is to synchronize the information of win between processes. When a RMA communication calls with argument win occur, there should be a synchronization call called on the win before, and another synchronization call to be called behind. The course is called an access epoch. MPI provides three synchronization mechanisms:

- The `MPI_WIN_FENCE` collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: all the process in the same communicator should synchronize.

- The four functions of `MPI_WIN_START`, `MPI_WIN_COMPLETE`, `MPI_WIN_POST` and `MPI_WIN_WAIT` can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize.
- Finally, shared and exclusive locks are provided by the two functions of `MPI_WIN_LOCK` and `MPI_WIN_UNLOCK`: the target process doesn't involve in the synchronization.

The synchronization operation of DMPI is similar to the barrier operation of the collective calls, but it does more works. During synchronization, every process's window information for communication is broadcast to other process involved in the communication. For the communication calls of RMA are all non-blocking, there should be a call do the polling operation to decide when and how to end the communication. In the point-to-point message-passing model, this call is `check_device`. In RMA, after the communication calls is done, we call the `check_device` function in the synchronization operation to finish the communication.

3.3 Dynamic Process Create and Management

We realized this functionality of Dynamic process management of DMPI in the DADI-E layer. The central works are done by RMS, which allocates and manages the dynamic process information. A balance loading can be provided with `mpirun` command when beginning to spawn new processes. This module can be divided into three parts: spawning new processes, establishing communication between two different applications, and resource control.

MPI users have two different functions to create new processes in runtime. The `MPI_Comm_spawn` function allows the creation of one or more process that will run the same executable with the same arguments. If the users wish to run different executables or to pass different arguments to the several processes they should use the `MPI_Comm_multiple_spawn` function. The spawning operation is collective over a certain intra-communicator. Only the process in the intra-communicator will be include in the inter-communicator that connect them to the new processes. A root process, indicated in the function's arguments, is responsible for actually creating the new processes. When one MPI's spawn function is executed, the new processes start their own MPI environment. Then the new environment and old environment are connected through the inter-communicator and users can exchange information between all processes.

MPI-2 provides functions to establish communication in a client/server model. The server process opens a MPI port and waits the client processes to connect to it. The MPI port is an implementation dependent entry point that allows the two type processes to exchange information about the two environments to create the inter-communicator between the processes.

Resource control is not included in the MPI-2 standard because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers.

In DMPI, the resource control for dynamic processes is something like that of PVM. There are two main tables to keep the process's information. They are GPT and CPT. The GPT records the global information of process, through which both the communication within processes and the management of processes can become easy. The information includes the global process id, the physic node id, and the port number of processes. Every communicator has a CPT that keeps the information of processes in the communicator. The information kept in CPT mainly includes the global rank, local rank in the communicator, and the index to the GPT. CPT is static table. The CPT is created as soon as a communicator is created. However, the GPT is a dynamic table. Once a process is created or updated, it is needed to change the GPT to keep the information up to date.

The works for resource control are mainly done with these two tables. These works include table creation, initialization, updating and so on. It is not a bad idea to let a daemon to do these works just like the implementation of Lam MPI and PVM.

4 Performance and Analysis

The tests are done on the testbed, which consists of 16 Linux workstations. Each node is a 2-way 1GHz PIII(Coppermine) SMP with 66MHz PCI Bus, and each one is running Turbo Linux 2.2.18-2smp. Two Myrinet M3S-PCI64B NICs are used on each node. The nodes are interconnected by M3S-SW16-8S switches.

The first figure shows the bandwidth of DMPI after optimized, compared with that of DMPI before optimized. Before optimization, the peak bandwidth is about 280MB/s when the data size is 128KB, which is only about 67% contrast to the bandwidth of BCL, 419MB/s. But the bandwidth of DMPI after optimized can reach 389MB/s, about 93% to that of BCL. The improvement is so obviously.

The second figure describes the latency of DMPI. It is about 16.3 us, while the BCL's latency is about 12us.

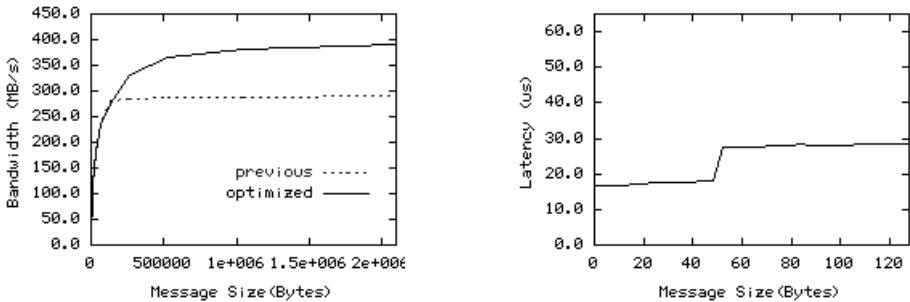


Fig. 3. Bandwidth and Latency of DMPI

5 Conclusion and Future Work

DMPI is ported from MPICH. Based on BCL, DMPI provides an efficient message-passing interface for a cluster for commodity SMPs. DMPI achieves 16.3 μ s of latency and 389MB/s of bandwidth for inter-node message passing.

With the characteristics of our system, we extended the ADI-2 to DADI-E that expands two modules of dynamic process management and remote memory access. Furthermore, DADI-E provides common APIs to both DMPI and DPVM. We also introduce an arithmetic of pipeline-sending to have the procedure of point-to-point communication optimized. After the optimization, DMPI now can have a very high performance, very closely to that of BCL.

At the future work, we plan to optimize the performance of collective operation. And another work is to do some research for MPICHG and MPI-2 standard. Furthermore, we plan to implement a high performance computing environment and grid environment.

References

- [1] Kai Hwang, Zhiwei Xu, Scaleable Parallel Computing: Technology, Architecture, Programming, WCB/McGraw-Hill, 1998.
- [2] Message Passing Interface (MPI) Forum Home Page: <http://www.mpi-forum.org>.
- [3] Message Passing Interface Forum: MPI-2: Extension to the Message-Passing Interface. (June 1997), available at <http://www.mpi-forum.org>.
- [4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing Vol.22, No.6, (September 1996).
- [5] W. Gropp and E. Lusk: MPICH Abstract Device Interface Version 3.3 Reference Manual Draft, Mar. 20, 2002.
- [6] MPICH Home Page: <http://www-unix.mcs.anl.gov/mpi/mpich>.
- [7] Xingfu Wu and Ninghui Sun: DPVM: PVM for Dawning Cluster Systems. Proc. of 4th International Conference/Exhibition on High Performance Computing in Asia-Pacific Region, 2000.
- [8] Ma Jie, He Jin, Meng Dan and Li Guo Jie: BCL-3: A High Performance Basic Communication Protocol for Commodity Superserver DAWNING-3000. Journal of Computer Science and Technology, 2001, 16(6): 522-530.
- [9] Meng Dan, Ma Jie, et al: Semi-User-Level Communication Architecture. In Proc. of IPDPS 2002, 2002.
- [10] Nanette J. Boden, etc.: Myrinet: A Gigabit-per-Second Local Area Network. IEEE Micro, Feb, 1995.
- [11] Myricom. The GM Message Passing System, <http://www.myri.com/GM/doc/>, 2000.

MPICH-CM: A Communication Library Design for a P2P MPI Implementation

Anton Selikhov, George Bosilca, Cecile Germain,
Gilles Fedak, and Franck Cappello

Laboratoire de Recherche en Informatique - CNRS and Université de Paris-Sud
Bâtiment 490, F-91405 Orsay Cedex, France
{selikhov,bosilca,cecile,fedak,fcj}@lri.fr

Abstract. The paper presents MPICH-CM – a new architecture of communications in message-passing systems, developed for MPICH-V – a MPI implementation for P2P systems. MPICH-CM implies communications between nodes through special Channel Memories introducing fully decoupled communication media. Some new properties of communications based on MPICH-CM are described in comparison with other communication architectures, with emphasis on grid-like and volunteer computing systems. The first implementation of MPICH-CM is performed as a special MPICH device connected with Channel Memory servers. To estimate the overhead of MPICH-CM, the performance of MPICH-CM is presented for basic point-to-point and collective operations in comparison with MPICH p4 implementation.

1 Introduction

Global Computing (GC) and Peer-to-Peer (P2P) [1] systems gather unprecedented processing power from borrowing time of idle computers. From the number of processor criterion, a GC system is thus an ideal infrastructure to run massively parallel applications. Nevertheless, only limited attempts have been done towards parallel programming on such systems.

Parallel execution models have been designed with a traditional machine model in mind, which can be summarized as strongly coupled: machines are reliable, and information flows reliably across computing entities (processes or threads). On the other hand, GC systems are extreme representatives of distributed systems. Failures are very frequent, for instance when a user reclaims his machine or unplugs his laptop. Failures are the worst case of faults, in that they are also perfectly (quite) unexpected: the machine simply disappears, leaving the system in whatever state it can be.

Message-passing in such a highly unreliable environment needs to address transparent virtualization as the basic issue: stable user-level MPI processes and logical communication endpoints must be implemented on top of ephemeral processes running in a GC system; a reliable communication protocol must be run on a fuzzy set of tasks. On top of this virtualization system, fault-tolerance

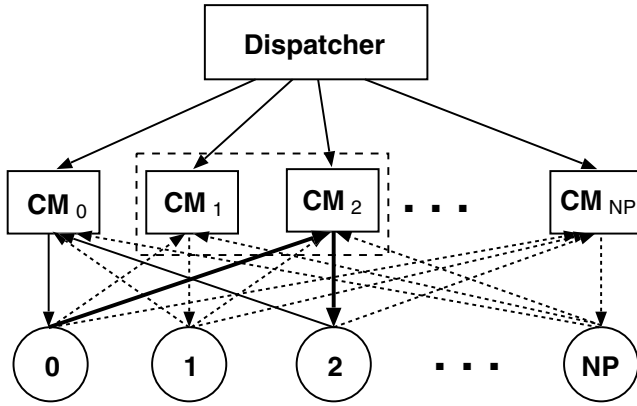


Fig. 1. MPICH-CM architecture components and their communication structure. Bold lines show a virtual channel from node 0 to node 2. The dashed box shows two Channel Memories hold by one CM server

can be achieved through execution rollback: tasks are checkpointed and lost ones are restarted from a process image saved in a stable storage [2].

All these issues are reflected in implementation of MPICH-V system [3], which the MPICH-CM communication library has been developed for. This paper presents only the virtualization layer: MPICH-CM, the communication layer of a GC parallel system. MPICH-CM is a full-fledged MPI implementation based on MPICH; it is highly distributed, thus is adapted to a P2P context, and fully asynchronous, being able to cope with non-existent target endpoints. Execution rollback, described in [3] is outside the scope of the paper; however, we will sketch in section 4 how the architecture allows for easy communication recovery.

2 MPICH-CM Architecture

MPICH-CM is based on the concept of communication tunneling, where a persistent communication channel is built on top of a variable architecture [4]. However, the variability tackled here is not at the level of transfer protocol, but at the level of endpoints. MPICH-CM mediates communication through *Channel Memories* (CM). The CMs act as proxies for sent messages, and repositories for the messages not yet delivered.

Fig. 1 describes the overall MPICH-CM architecture. The Dispatcher is responsible for mapping 1) the global MPI objects (parallel applications and Communicators), to sets of tasks running on Nodes, 2) Channel Memories servers to Nodes. The last mapping is *destination oriented*: each Node is associated with one CM, called its *owner CM*, which stores the messages en route to this Node.

Channel Memory servers handle communication requests, actually decoupling the communicating nodes. Besides, whatever may be the firewall in front

of the nodes, they only need to communicate with a CM Server. While a parallel node may fail unexpectedly, CM servers are expected to be stable, therefore always reachable (meaning that a failure is fatal). Currently, this is implemented through dedicated servers; whether the CM service should be provided by volatile resource, or kept for stable ones, it is an instance of the general open question of flat vs hierarchical internal organization for P2P systems [5].

The Dispatcher distributes CMs to CM servers at the initialization time through an Application Channel Memory Map (ACMMap). The ACMMap actually translates MPI Communicators into CM ownership: one CM Server needs only to know about its owned nodes, and the nodes which may send messages to one of its owned nodes; thus, the CM Server in-degree is determined by its out-degree (number of owned nodes), which is a decision of the Dispatcher, and by the communication topology of the application. For instance, a 5-points stencil application (2D grid topology) gives a in-degree bounded above by 4 times the number of owned nodes (in-degree). On the contrary, Fig. 1 shows the CM communication structure associated to a fully connected application topology: each node connects to all CMs, as it has to communicate with all the other nodes; as stated before, only one CM is input for any node. Knowing the communication requirements of an application, for instance from its Communicators, the Dispatcher can tune the resources (storage and server nodes) allocation accordingly.

3 The Node Library

MPICH-CM is nearly transparent to the user: the only requirement is to link with the MPICH-CM library, which is in charge of interfacing with the Channel Memory system. The MPICH-CM library is built in the regular MPICH manner, through a device implementing the Chameleon Interface functions *PIbsend*, *PIbrecv* (for blocking communications with Channel Memory), *PInprobe*, *PIfrom*, *PIInit* and *PIFinish*. This *ch_CM* device is thus at the same level as *ch_p4*.

3.1 Initialization and Finalization

When a parallel application is dispatched, each MPI process calls the *ch_CM* implementation of the *PIInit* function. *PIInit* connects to the owner CM Server, and receives the ACMMap. The next step is synchronization: before starting to communicate, the region of the CM set that the node will have to deal with must be ready. Thus, the node registers to all its destination CMs, resulting in resources allocation at the CM level, and creation of connections. *PIInit* returns only when the node has received acks from all CMs it has registered to.

The connections established with CM servers during this phase are TCP sockets open until the application finishes communication by calling *MPLFinalize*.

At the user level, *MPLFinalize* notifies all processes about the end of the MPI communication structure. The device-dependent *PIFinish*, which task is to shut down the device, sends a system message (see next subsection) to all CM servers to notify them and closes all corresponding sockets.

3.2 Communication

The implementation of all communication functions follows the same scheme. First, a *system message* notifies a CM about the type of communication and the properties of the message to be sent or received (*source*, *destination*, *size*, *tag* and *kind*). Next, the message body (PIx buffer) is sent or received. System messages are used by CM Servers 1) to select the appropriate handler for the next request 2) to know about its parameters. We named them System and not Control messages, to be not confused with MPICH high level Control messages.

All the communications with the CM server are passed through blocking *write* and *read* on UNIX sockets.

PIbsend copies its *tag*, *length* and *to* parameters to the corresponding fields of a system message, fills *source* information with its MPI rank and sends the message to a CM server, which is selected according to the destination rank using the ACMMMap. The message body follows using the *buffer* parameter of the *PIbsend*.

PIbrecv uses the *tag* parameter to determine the source of the message to be received (when the source is important and for non-rendezvous protocol of communications), sets the *tag* and *length* fields and sends it to the owner CM server. Next, it waits to receive a system message to know about the source and the size of the body, which is received immediately after this.

PIfrom returns the value of the *source* field of the last system message received by *PIbrecv*.

PIprobe uses the *tag* parameter and the rank number to fill the corresponding fields of a system message. It sends the message to the owner CM and receives information about existence of messages in the CM.

4 The Channel Memory Architecture

A CM is basically a data structure devoted to message storage. A CM Server interfaces one or multiple CMs to other components (Nodes and Dispatcher). The main functions of a CM Server include accepting requests to handle new applications from a Dispatcher, accepting connections from the nodes involved in the application and managing all communications (handling multiple CMs). To make the best use of the available bandwidth, a CM Server is multi-threaded. A pool of thread is allocated at init time, and at each instant, only one thread is waiting on *select* for any activity on all known sockets. Due to lack of space, we focus on communication handling, skipping all init phases.

An essential property of the Channel Memory protocol is existence of intermediate message queues. This differs from the p4 device, where all messages are retrieved from a channel and queued locally. Therefore, the protocol involves two overheads: queue management and doubling the number of communications. While the second one cannot be avoided, the first one is decreased by special data structures, and tuned message storing and retrieving mechanisms.

4.1 Data Storage Structures

According to MPICH specifications [6], passing any user data breaks down into Control Message(s) and, maybe, a Data message, depending on the actual size of the user data (very short messages are included in the Control message body). A CM Server stores these two types of MPICH messages, for each owned node, in a separate Control queue, and an array of data queues, which has n queues, where n is the whole number of nodes in the communication group. An index of a data queue in the array corresponds to the rank of source of these data messages. Both Control queue and Data queue are organized as FIFO. This organization allows to keep the time ordering of messages issued from a particular node and to retrieve Data messages by source rank. The first property means that all the messaging events are fully logged (data and temporal ordering), allowing communication replay [3]. The combination of the two properties provides constant time data message retrieval.

4.2 Queues Management

Handling of request to communication depends on the *kind* of request received in a system message.

The *PInprobe* handler tests the Control queue corresponding to this node and sends the result.

Upon a request to *receive* a Control message, the first message (if exists) is retrieved from the Control queue and sent to the node; otherwise the node is marked as "waiting for a Control". Receiving Data message includes an attempt to retrieve the first data message in the queue according to the requested source and, like in the case of Control messages, leads to either sending the data message found to the node, or marking the node as "waiting for a Data".

Upon a request to *send* a message, whether Data or Control, to the owned destination node, the message is first buffered. Next, if the destination client is "waiting for a Control (Data)", the message is directly sent to the destination node; otherwise it is stored in the Control (Data) queue.

5 Performance Evaluation

Performance tests results were obtained for both p4 and CM- based MPICH on the LRI cluster of PCs (500 to 700MHz), connected through a switch with 100 Mbit/s maximal throughput for each node. The aim of the tests was to explore the overhead involved by the CM architecture and the architecture scalability. All measurements, except the last one, are averages over one hundred runs.

5.1 RTT Performance

First, the round-trip time (RTT) for blocking communication was measured for two nodes, communicating through two CM servers (Fig.2, left graph), which is

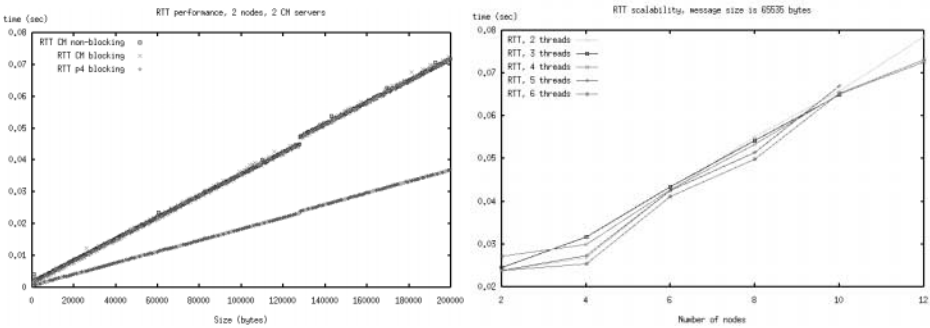


Fig. 2. RTT vs message size (left graph) and number of nodes on one CM (right graph)

the best situation (one CM server per Node). MPICH-CM reaches nearly half of the performance of the p4 based MPICH. Moreover, the linear behavior and the slope show that the performance is strictly determined by the underlying TCP bandwidth. Thus, the overhead incurred in the CM server itself is negligible.

Second, the scalability of a CM Server was investigated on the base of RTT measurements (Fig.2, right graph). In this test, only one CM Server was used to manage all CMs. The nodes were divided in pairs, each pair communicating independently. The quasi-linear behavior above 4 nodes follows the bandwidth limitation. However, the threaded architecture helps for low in-degree: from 2 to 4 nodes, and 4 threads, the latency is nearly constant.

5.2 MPIAlltoall Performance

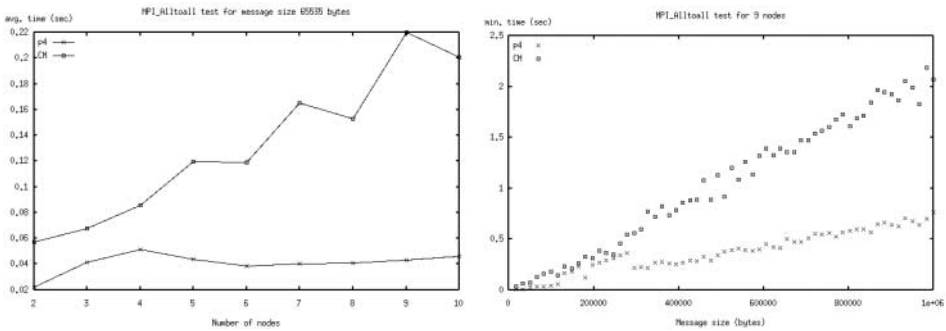


Fig. 3. MPIAlltoall vs number of nodes managed by one CM server (left graph), and vs message size (right graph)

Estimation of *MPI_Alltoall* time is used often for performance analysis of multiprocessor systems and clusters. It involves very intensive communications and allows to estimate the scalability of the system being tested. For MPICH-CM based system, the effect of message size and number of nodes was measured. In both cases, only one CM Server was used to manage all CMs. The reported times are measured on node 0, with no special synchronization between consecutive calls to the Alltoall routine, besides the one involved by the operation itself.

The increasing times for *MPI_Alltoall* operation presented on Fig. 3, left graph, has two reasons: first, the overhead of message transition through a CM, illustrated in the first experiment on Fig. 2 and second, the overhead of managing all the nodes by only one CM server, which is illustrated in the Fig. 2.

Fig. 3 (right graph) presents the results of measurement of *MPI_Alltoall* minimal time for 9 nodes with increasing size of message being sent. For the same reason of having only one CM server for managing all CMs, the increase of time for this operation is faster than in the case of RTT measurements.

6 Related Work

The Channel Memory approach has two motivations: communication decoupling and communication events logging. The first one has been pioneered by Linda [7]. Besides that, most of Linda features are related to a high-level parallel programming language, based on a shared-memory abstraction for interprocess communications, implemented through associative search, while MPICH-CM is a pure message-passing system with named communicating entities. The project closest to MPICH-V is MPI-FT [8]. It uses a monitoring process, the observer, for providing MPI level process failure detection and recovery mechanism based on message logging. Our work is a step more in the same direction, as it merges communication decoupling and logging. Many other works [9, 10], tackles the issue of fault-tolerance by exposing the faults and the computation state at the application level. Yet another way has been explored by the Condor team [11], by global distributed checkpointing following the Lamport algorithm.

7 Conclusion and Future Work

This paper has presented MPICH-CM, a communication library designed for MPICH-V, a MPI implementation for GC and P2P systems. The experimental results have shown 1) predictable and explainable limitations in its performance; 2) scalability if one is willing to provide enough resource for communication, that is enough CM Servers. This may appear costly; however, a network of volatile or unreliable nodes may become a more common substrate when considering very large clusters, or clusters or clusters, running during quite long computing time. In this case, it may be more efficient to allocate resources for ordered logging of communication events, which is the core of the CM architecture, than to restart execution of all nodes many times from the very beginning and to hope successful finalizing. Other salient features of MPICH-CM are:

- Firewall bypass. All communications between nodes are initiated by a node and addressed to a specialized port of a CM Server, not directly to a peer. Thus, both communicating peers may stay behind firewalls
- Effective pipelining. Channel Memories may be used for the creation of an effective pipelined system by forwarding on the fly messages on their way from one node to another. This may be useful for some specialized video-processing systems.
- Fully asynchronous communication. Neither the sending nor the receiving node is required to wait until the end of the communication. All communication requests are absorbed by the Channel Memories, allowing nodes with different communication throughput to work with maximal performance.

We are currently integrating MPICH-CM inside the XtremWeb Global computing platform [12, 13] developed at LRI, together with a checkpoint and recovery facility. The complete system will provide a framework for transparent parallel execution of MPI programs on volunteer-based resources.

References

- [1] Oram, A. (ed.): P2P: Harnessing the Power of Disruptive Technologies. O'Reilly (2001) 323
- [2] Elnozahy, E., Jonhnson, D., Wang, Y.: A Survey of Rollback Recovery Protocols in Message-Passing Systems. CMU TR-96-181. Carnegie Mellon University (1996) 324
- [3] Bosilca, G. et al.: MPICH-V: Parallel Computing on P2P systems. To appear in IEEE-ACM SC2002: High Performance Networking and Computing (2002) 324, 327
- [4] Al-Khayatt, S. et al.: A Study of Encrypted, Tunneling Models in Virtual Private Networks. 4th IEEE Conf. in IT and computing & coding (2002) 324
- [5] Kan, G.: Gnutella. In: Oram, A. (ed.): P2P: Harnessing the Power of Disruptive Technologies. O'Reilly (2001) 325
- [6] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing, Vol. 22, (1996) 789-828 327
- [7] Bakken, D.E., Schlichting, R.D.: Supporting Fault-Tolerant Parallel Programming in Linda. IEEE Trans. on Paral. and Distrib. Systems, Vol. 6(3), (1995) 287-302 329
- [8] Louca, S. et al.: MPI-FT: a Portable Fault Tolerant Scheme for MPI. Parallel Processing Letters, Vol.10(4). World Scientific, New Jersey London Singapore Hong Kong. (2000) 371-382 329
- [9] Fagg, G., Bukovsky, A., Dongarra, J.: Harness and Fault Tolerant MPI. Parallel Computing. North-Holland. Vol. 27(11), (2001) 1479-1479 329
- [10] Agbaria, A., Friedman, R.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. 8th IEEE Int. Symp. on High Perf. Dist. Comp. (1999) 329
- [11] Pruyne, J., Livny, M.: Managing Checkpoints for Parallel Programs. Workshop on Job Scheduling Strategies for Parallel Processing, IPPS'96. IEEE Press. (1996) 329

- [12] Fedak, G., Germain, C., Neri, V., Cappello, F.: Xtremweb: A Generic Global Computing Platform. IEEE/ACM CCGRID'2001. IEEE Press. (2001) 582-587 [330](#)
- [13] Germain, C., Fedak, G., Neri, V., Cappello, F.: Global Computing Systems. 3rd Int. Conf. on Scale Scientific Computations, Lecture Notes in Computer Science, Vol.2179. Springer-Verlag, Berlin Heidelberg New York. (2001) 218-227 [330](#)

Design and Implementation of MPI on Portals 3.0

Ron Brightwell¹, Arthur B. Maccabe², and Rolf Riesen¹

¹ Scalable Computing Systems, Sandia National Laboratories*
Albuquerque, NM 87185-1110, USA
{bright,rolf}@cs.sandia.gov

² Computer Science Department, University of New Mexico
Albuquerque, NM 87131-1386
maccabe@cs.unm.edu

Abstract. This paper describes an implementation of the Message Passing Interface (MPI) on the Portals 3.0 data movement layer. Portals 3.0 provides low-level building blocks that are flexible enough to support higher-level message passing layers such as MPI very efficiently. Portals 3.0 is also designed to allow for programmable network interface cards to offload message processing from the host processor. We will describe the basic building blocks in Portals 3.0, show how they can be put together to implement MPI, and describe the protocols of an MPI implementation. We will look at several key operations within an MPI implementation and describe the effects that a Portals 3.0 implementation has on scalability and performance.

1 Introduction

The advent of cluster computing has motivated much research into message passing APIs and protocols targeted for delivering low-latency high-bandwidth performance to parallel applications. Relatively inexpensive programmable network interface cards (NICs), like Myrinet [1], have made low-level message passing protocols and programming interfaces a popular area of research [17, 18, 13, 11, 7, 4, 14]. Most of these activities have been focused on delivering latency and bandwidth performance as close to the hardware limitations as possible.

The current Portals [3, 2] data movement interface (Portals 3.0) is an evolution of networking technology initially developed for large-scale distributed memory massively parallel systems. Portals began as a key component of our lightweight compute node operating systems [9, 16], and has evolved into a programming interface that can be implemented efficiently for different operating systems and networking hardware. In particular, Portals provides the necessary building blocks for higher-level protocols to be implemented on programmable

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

or intelligent network interfaces without providing mechanisms that are specific to each higher-level protocol. This paper describes how these building blocks and their associated semantics can be combined to support the protocols needed for a scalable high-performance implementation of the Message Passing Interface (MPI) Standard [10].

The rest of this paper is organized as follows: In the next section, we give a brief overview of the Portals 3.0 API. In section 3, we present the initial implementation of MPI on Portals 3.0. Section 4 describes a problem that we encountered with this implementation and Section 5 describes a second implementation of MPI that uses a new semantic added to Portals to overcome this limitation. We briefly discuss the benefits of our implementation with respect to progress in Section 6 and conclude in Section 7 with summary of the key points of this paper.

2 Portals 3.0

The Portals 3.0 API is composed of elementary building blocks that can be combined to implement a wide variety of higher-level data movement layers such as MPI. We have tried to define these building blocks and their operations so that they are flexible enough to support other layers as well as MPI, but MPI was our main focus. The following sections describe Portals objects and their associated functions.

The Portals library provides a process with access to a virtual network interface. Each network interface has an associated Portal table that contains at least 64 entries. The table is simply indexed from 0 to $n - 1$, and the entries in the table normally correspond to a specific high-level protocol. Portal indexes are like port numbers in Unix. They provide a protocol switch to separate messages intended for different protocols.

Data movement is based on one-sided operations. Other processes can use a Portal index to read (get) or write (put) the memory associated with the portal. Each data movement operation involves two processes, the **initiator** and the **target**. In a put operation, the initiator sends a put request containing the data to the target. The target translates the Portal addressing information using its local Portal structures. When the request has been processed, the target may send an acknowledgement. In a get operation, the initiator sends a get request to the target. The target translates the portal addressing information using its local Portal structures and sends a reply with the requested data.

Typically, one-sided operations use a triple to address remote memory: $\langle \text{process id, buffer id, offset} \rangle$. In addition, portal addresses include a set of match bits. Figure 1 presents a graphical representation of the structures used to translate Portal addresses. The process id is used to route the message to the target node. The buffer id is used as an index into the Portal table, this identifies a match list. The match bits (along with the id of the initiator) are used to select a match entry (ME) from the match list. The match entry identifies a list

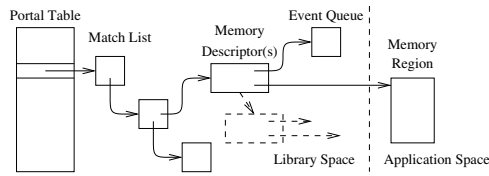


Fig. 1. Portal Addressing Structures

of memory descriptors. The first of these identifies a memory region and an optional event queue.

The match entries in a match list are searched in sequential order. If a match entry matches the request, the first memory descriptor associated with the match entry will be tested for acceptance of the message. If the match entry does not match the message or the memory descriptor rejects the message, the message continues to the next match entry in the list. If there are no further match entries, the message is discarded.

Memory descriptors (MDs) have a number of options that can be combined to increase their utility. They can be configured to respond to put operations, get operations, or both. Each memory descriptor has a threshold value that determines how many operations can occur before the descriptor becomes inactive. In addition, each memory descriptor has an offset value which can be managed locally or remotely. When it is managed locally, the offset is increased by the length of each message that is deposited into the memory descriptor. Consecutive messages will be placed in the user's memory one after the other. Memory descriptors can specify that an incoming message which is larger than the remaining space will be truncated or rejected.

Memory descriptors can be chained together to form a list. Moreover, each memory descriptor can be configured to be unlinked from the list when it is consumed (i.e., when its threshold becomes zero). When the last descriptor in the list is consumed and unlinked, the associated match entry also becomes inactive. Each match entry can be configured to unlink when it becomes inactive.

The decision to generate an acknowledgement in response to a put operation requires input from both the initiator and the target. The initiator must request an acknowledgement and the memory descriptor used in the operation must be configured to generate an acknowledgment.

Memory descriptors may have an associated event queue (EQ). Event queues are used to record operations that have occurred on memory descriptors. Multiple memory descriptors can share a single event queue, but a memory descriptor may only have one event queue.

There are five types of events that represent the five operations that can occur on a memory descriptor. A `PTL_EVENT_GET` event is generated when a memory descriptor responds to a get request. A `PTL_EVENT_PUT` event is generated when a memory descriptor accepts a put operation. A `PTL_EVENT_SENT` event is gener-

ated when it is safe to manipulate the memory region used in a put operation. A `PTL_EVENT_REPLY` event is generated when the reply from a get operation is stored in a memory descriptor. A `PTL_EVENT_ACK` event is generated when a acknowledgement arrives from the target process. In addition to the type of event, each event records the state of the memory descriptor at the time the event occurred.

3 Initial MPI Implementation

In this section, we describe our MPI implementation for Portals 3.0. This implementation is a port of MPICH [6] version 1.2.0, which uses a two-level protocol based on message size to optimize latency for short messages and bandwidth for large messages. In addition to message size, the different protocols are used to meet the semantics of the different MPI send modes.

We use the match bits provided by Portals to encode the send protocol (3 bits), the communicator (13 bits), the local rank (within the communicator) of the sending process (16 bits) and the MPI-tag (32 bits). During `MPI_Init()`, we set up three Portal table entries. The *receive Portal* is used for receiving MPI messages. The *read Portal* is used to for unexpected messages in the long message protocol. The *ack Portal* is used for receiving acknowledgements for synchronous mode sends. We also allocate space for handling unexpected messages, the implementation of which we describe below. After these structures have been initialized, all of the processes in the job call a barrier operation to insure that all have been initialized.

3.1 Short Message Protocol

Like most MPI implementations, we use an eager protocol for short messages (standard and ready sends). The entire user buffer is sent immediately and “unexpected messages” (where there is no pre-posted receive) are buffered at the receiver.

For standard sends, we allocate an MD to describe the user buffer and an EQ associated with the MD. We configure the MD to respond to put operations and set the threshold to one. We create an address structure that describes the destination and fill in the match bits based on the protocol and MPI information. Using the Portal put function, we send the MD to the *receive Portal* of the destination process, requesting that no acknowledgement be sent. This send is complete when an event indicating that the message has been sent appears in the EQ.

For synchronous sends, we need to know when the message is matched with a receive. We start by allocating an MD and an EQ as described earlier. Next, we build an ME that uniquely matches this message. This ME is associated with the MD allocated in the previous step and attached to the local *ack Portal*. We configure the MD to respond to acknowledgements and put operations and set the threshold to two. When the Portal put operation is called, we request an

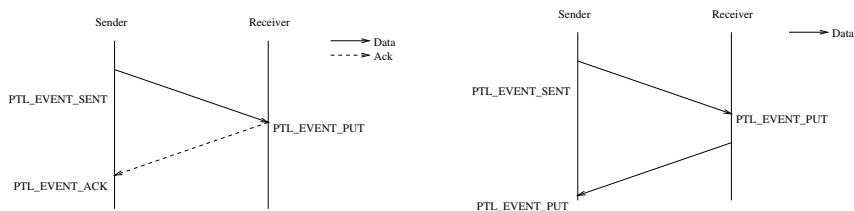


Fig. 2. Short Synchronous Send Protocol: Expected and Unexpected

acknowledgment, and include the match bits for the ME in 64 bits of out-of-band data, called the header data.

Completion of a short synchronous mode send can happen in one of two ways. If the matching receive has been posted, an acknowledgment is generated by the remote memory descriptor as illustrated in left side of Figure 2. If the matching receive is not posted, the message is buffered until the matching receive is posted. At this point, the receiver will send an explicit acknowledgment message using the Portal put operation, as illustrated on the right side of Figure 2. A short synchronous mode send completes when the `PTL_EVENT_SENT` event has been recorded and either the `PTL_EVENT_ACK` event or the `PTL_EVENT_PUT` event has been recorded.

3.2 Long Message Protocol

Unlike most MPI implementations, we also use an eager protocol for long messages. That is, we send long messages assuming that the receiver has already posted a matching receive. Because the message could be discarded, we also make it possible for the receiver to get the message when it finally posts a matching receive.

We start by inserting an ME that uniquely describes this send on the *read Portal*. We create an MD that describes the user buffer. This MD is configured to respond to put operations, get operations, and ack operations. Since all of three of these may occur, we set the MD's threshold to three. We then attach the MD to the ME. We set the protocol bits in the match bits for a long send and fill in the other match bits appropriately. We call the Portal put function, requesting an acknowledgment and we include the match bits of the ME in the header data.

As with the short synchronous mode sends, the long send protocol can complete in one of two ways. First, if the message is expected at the receiver, an acknowledgment is returned. In this case, the event queue will contain a `PTL_EVENT_SENT` event and a `PTL_EVENT_ACK` event. After these two events have been recorded, the send is complete. This is illustrated on the left side of Figure 3.

If the message was not expected, an acknowledgment is returned to the sender indicating that the receiver accepted zero bytes. The sender must then wait for the receiver to request the data from the sender's read Portal. In this

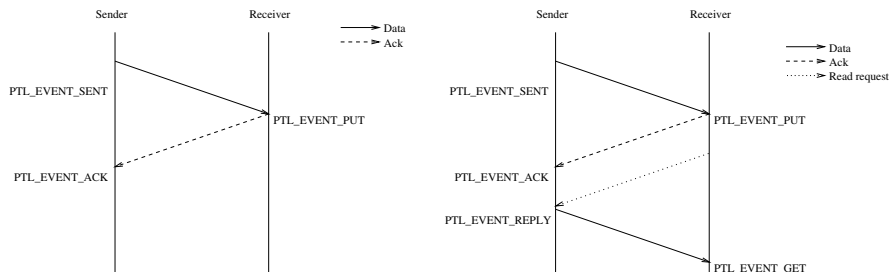


Fig. 3. Long Standard Mode Send: Expected, Unexpected

case, three events mark the completion of the send: a `PTL_EVENT_SENT` event, a `PTL_EVENT_ACK` event, and a `PTL_EVENT_GET` event. This is illustrated on the right side of Figure 3.

Since the completion of this send protocol is dependent on a matching user buffer being posted at the receiver, this protocol is the same for standard mode and synchronous mode sends. Moreover, the ready mode send for long messages is identical to a short standard mode send. Since the MPI semantics guarantee that a matching buffer is posted at the receiver, the sender need not wait for an acknowledgment or set up an entry on the read Portal.

3.3 Posting Receives

Posting a receive involves two lists: the posted receive list and the unexpected message list. The unexpected message list holds messages for which no matching receive has been posted. Before a receive can be added to the posted receive list, we must search the unexpected list. Figure 4 illustrates the match list structure we use to represent these two list. This list starts with entries for the posted receives (no entries are shown), followed by a *mark* entry that separates the two lists, followed by entries for unexpected messages.

When a process calls an MPI receive, we create a memory descriptor that describes the user buffer. The MD is configured to accept put operations, generate acknowledgements, unlink when used, and is given an initial threshold of zero (making the MD inactive). The communicator id, source rank within the communicator, and message tag are translated into match bits. We use these match bits along with the MD to build an ME which is inserted just in front of the *mark* entry in the match list.

Next, we search the unexpected messages for a match. If no match is found we activate the MD by setting its threshold to one (this test and update is atomic with respect to the arrival of additional unexpected messages). If we find a match, we unlink the ME from the match list and take the appropriate action based on the protocol bit in the message. If the message is a long protocol message, we activate the MD and perform a Portal get operation, which reads the send buffer from the sender. If the message is a short protocol message, we simply

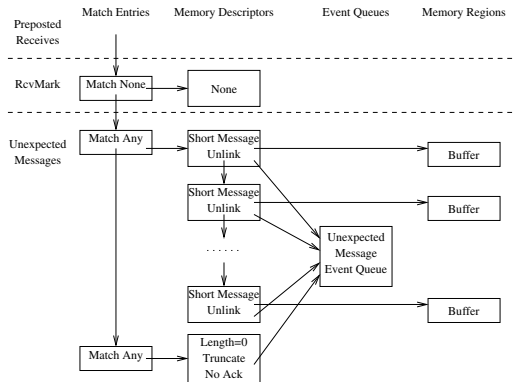


Fig. 4. Message Reception in the Initial Implementation

copy the contents of the unexpected buffer into the user's receive buffer. If the header data in the event is nonzero, this indicates that the send is synchronous. In this case, we send a zero-length message to the ack Portal at the sender using the Portal put operation.

4 Limitations

When the MPI library is initialized, we create two match entries, one for unexpected short protocol messages and another for unexpected long protocol messages. The match entry for short messages will match any message that has the short message bit set. There are 1024 memory descriptors attached to this match entry. Each MD provides an 8 KB memory buffer, has a threshold of one, is configured to accept put operations, and does not automatically generate an acknowledgment. The match entry for long unexpected messages has an MD of zero bytes with an infinite threshold that is configured to accept and truncate put operations and automatically generate an acknowledgment. All of the MD's for unexpected messages are attached to the same event queue to preserve message ordering. This event queue is created with 2048 entries.

Despite being able to support 1024 outstanding unexpected short messages at any time, this limitation proved to be too restrictive in practice, especially as applications scaled beyond 700 processes. Our application developers typically think of limits in terms of the messages sizes rather than message counts. In our implementation, an unexpected message of 0 bytes would consume an 8 KB memory descriptor. To an application developer, this message shouldn't consume any buffer space at all. Moreover, in a NIC-based implementation, 1024 MD's consumes a significant amount of a limited resource, NIC memory, leaving fewer MD's for posted receives.

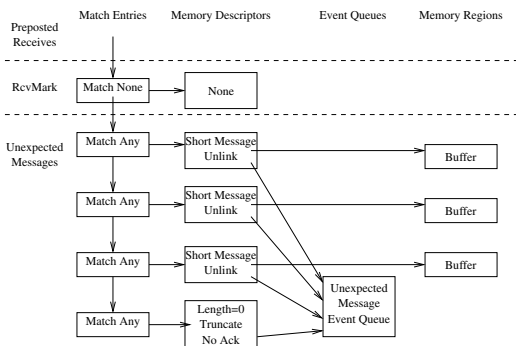


Fig. 5. Message Reception in Second Implementation

5 Second Implementation

To address these problems, an additional threshold semantic was added to MDs. An MD could be created with a maximum offset, or high-water mark. Once this offset was exceeded, the MD would become inactive. Figure 5 illustrates our new strategy for handling short protocol unexpected messages. We now create three match entries for unexpected short messages, all with identical selection criteria. Each ME has an MD attached to it that describes a 2 MB buffer. As unexpected messages come into the MD, they are deposited one after the other until a message causes the maximum offset to be exceeded. When this happens, the MD is unlinked, and the next unexpected short message will fall into the next short unexpected MD. Once all of the unexpected messages have been copied out of the unlinked MD, the ME and MD can be inserted at the end of the short unexpected ME's.

This new strategy allows for the number of unexpected messages to be dependent on space rather than count. It also reduces the number of MD's for short unexpected messages to three, significantly reducing the amount of NIC resources required by MPI. The handling of posted receives and unexpected long protocol messages did not change. This semantic change to Portals eliminated the need for lists of memory descriptors, so the API was changed to allow only a single memory descriptor per match entry.

6 Progress

Unlike most MPI implementations described in current literature [8, 15, 12, 5], the semantics of Portals 3.0 support the necessary progress engine for an MPI implementation without the need for explicit application intervention. Portals' flexible message selection semantics and the use of eager protocols allow communication operations to make progress independent of making frequent MPI library calls. While progress of Portals messages is dependent on the underlying transport layer, NIC-based implementations are able to support progress

for Portals, and hence MPI, without any other mechanism, such as a user-level thread.

7 Summary

This paper has described an implementation of MPI on the Portals 3.0 data movement layer. We have illustrated how Portals 3.0 provides the necessary building blocks and associated semantics to implement MPI very efficiently. In particular, these building blocks can be implemented on intelligent network interfaces to provide the necessary protocol processing for MPI without being specific to MPI. The implementation described in this paper has been in production use on a 1792-node Alpha/Myrinet Linux cluster at Sandia National Laboratories for more than two years.

References

- [1] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-A Gigabit-Per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995. 331
- [2] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002. 331
- [3] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999. 331
- [4] Compaq, Microsoft, and Intel. Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997. 331
- [5] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In *Proceedings of the Third MPI Developers' and Users' Conference*, pages 15–24, March 1999. 338
- [6] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. 334
- [7] Y. Ishikawa, H. Tezuka, and A. Hori. PM: A High-Performance Communication Library for Multi-user Parallel Environments. Technical Report Technical Report TR-96015, RWCP, 1996. 331
- [8] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997. 338
- [9] A. B. Maccabe, K. S. McCurley, R. E. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.*, pages 245–251, June 1994. 331
- [10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994. 332
- [11] Myricom, Inc. The GM Message Passing System. Technical report, Myricom, Inc., 1997. 331

- [12] F. O'Carroll, A. Hori, H. Tezuka, and Y. Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *ACM SIGARCH ICS'98*, pages 243–250, July 1998. 338
- [13] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages(FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. *IEEE Concurrency*, 40(1):4–18, January 1997. 331
- [14] L. Prylli. BIP Messages User Manual for BIP 0.94. Technical report, LHPC, June 1998. 331
- [15] L. Prylli, B. Tourancheau, and R. Westrelin. The Design for a High Performance MPI Implementation on the Myrinet Network. In *Proceedings of the 6th European PVM/MPI Users' Group*, pages 223–230, September 1999. 338
- [16] P. L. Shuler, C. Jong, R. E. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceedings of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995. 331
- [17] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP*, pages 40–53, December 1995. 331
- [18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communications and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992. 331

Porting PVM to the VIA Architecture Using a Fast Communication Library

Roberto Espenica¹ and Pedro Medeiros²

¹ Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Beja
r.espenica@estig.ipbeja.pt

² Departamento de Informática, Universidade Nova de Lisboa
pm@di.fct.unl.pt

Abstract. In this paper we present an implementation of PVM over the VIA architecture. Using VIA, the PVM communication primitives performance approaches the real hardware capabilities. As VIA is an industry standard for high performance communication on system area networks, this implementation runs on every VIA-conformant platform. The current PVM implementation is based on Berkeley Sockets. To ease the integration of VIA, a stream library (LSL - Lite Stream Library) was developed. LSL supplies a socket-like interface to VIA. LSL can be used directly by applications, thus improving their communication performance specially for small messages. Performance results obtained in the current prototype (Linux cluster using M-VIA) already show that LSL has some performance gains over the native socket interface, but is still open to enhancements.

1 Introduction

Cluster environments for parallel computing use off-the-shelf low latency and fast Local Area Networks (e.g. Fast or Gigabit Ethernet). In this environment, the default is to use the standard transport and network Internet protocols (IP, TCP, UDP). This has the consequence that not all the communication hardware bandwidth is achieved. TCP/IP has a poor performance for small messages, that are quite common in many parallel applications. Previous work [1] has shown that most overheads in traditional communication systems are caused by software, e.g., system calls, buffer management, and data copying. To avoid these costs, several lightweight communication layers have been proposed [1, 3] which eliminate system calls and reduce or eliminate buffer management overheads.

1.1 Lightweight Libraries

The objective of a lightweight library is performance. The application of these libraries will be mainly on parallel applications. There are well known factors in standard communication systems that have a negative impact on performance. Modern LANs have very low error rate, so it is possible to build an optimistic protocol that assumes no errors and due to its simplicity have small impact on

performance. In case of error the library informs the application and does not try to recover from error.

Multiple copy operations between application memory space and communication system memory space have a huge impact on system performance. An ideal system transmits data directly to/from application memory. These systems implement zero-copy protocols. There are several solutions, like locking the user data structures into physical RAM and let the NIC access them directly via DMA upon communication. Other solution is to remap the kernel-level temporary buffers into user memory space. These buffers are accessed directly by the NIC when sending/receiving data.

System calls also introduce overhead. The switch from user-level to kernel-level and again to user-level invoking the scheduler before re-entering user-level has an impact in performance that's not neglectable. One solution is to implement the communication system entirely at user level. The buffers and registers of the NIC are remapped from kernel space into user memory space so that user processes no longer have to cross the user-kernel boundary to drive the device. This is the basic idea of the so-called user-level communication architecture. An alternative to system calls are lightweight system calls. These system calls save a subset of CPU registers and do not invoke the scheduler before reentering user-level.

1.2 Virtual Interface Architecture

Using the principles suggested above, an industry standard for lightweight communication targeted to system area networks called Virtual Interface Architecture (VIA) was developed. Communication libraries that use the VIA [2] interface will run on every VIA-conformant hardware. The VIA specification describes a software interface for fully protected, user-level access to a network hardware. Network interface cards (NICs) can be designed to support the VIA specification at the hardware level for additional performance acceleration.

Figure 1 depicts the organization of VIA and its four basic components: Virtual Interfaces (VIs), Completion Queues (CQs), VI Providers, and VI Consumers. Each application (VI Consumer) that uses the VIA has a protected interface to network hardware, that is called VI. A VIA connection has a VI in each of the end-points, that is, for each connection there are two VIs. Each connection is a bi-directional and point-to-point communication path. A VI consists of a pair of Work Queues: a send queue and a receive queue. The application post requests, in the form of descriptors, on the Work Queues to send or receive data. A descriptor is a memory structure that contains all of the information that the VI Provider needs to process the request, such as pointers to data buffers. All the memory regions used for communication should be registered prior to submitting the request, so that the regions are pinned into the physical memory during data transfer. This is because VIA allows the network adapter to read and write data directly from and to the user address space, thus enabling the zero-copy protocol.

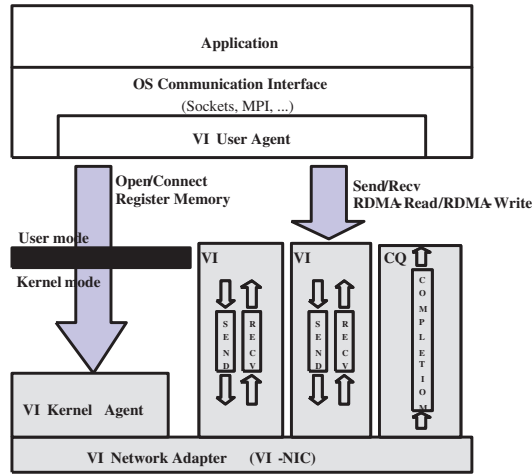


Fig. 1. Virtual Interface Architecture

1.3 Overview of the Paper

In section 2 we discuss the problems that arise when porting PVM to new communication architectures and outline the solutions used in our implementation, namely the use of an intermediate socket-like layer. Section 3 describes this last layer - LSL (Lite Stream Library), presents performance results, and shows how LSL is used in the PVM port to VIA. Finally, in section 4 we present the conclusions of this work and outline the future work in enhancing the performance of LSL.

2 Porting PVM to VIA

In this section we begin by describing PVM internal architecture, in order to justify the options we used in the port of PVM to VIA.

2.1 PVM Internal Architecture

PVM was developed on TCP/IP using the sockets interface [5]. Both TCP and UDP transport protocols are used. When UDP is used, and due to its characteristics, a small layer of software is added to get a protocol of reliable delivery of messages (using ACKs, Retries, timeouts) that guarantees the sending order and eliminates repeated messages. Pvmmd communicates with all the others pvmmds and with the local tasks. There is no communication between a pvmmd and remote tasks. The TCP/IP protocols used for the communication between pvmmds and tasks are different. Between pvmmds the protocol used is UDP plus the above-mentioned layer. Between pvmmds and tasks the protocol used is TCP.

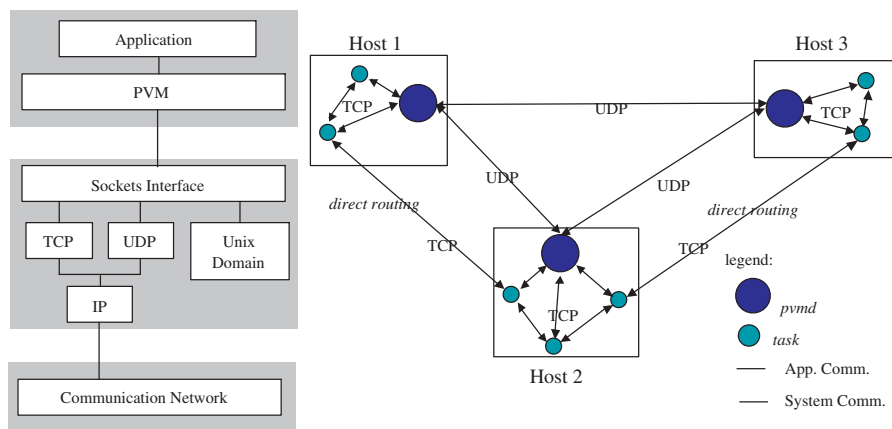


Fig. 2. Internal communication protocols used in PVM

All communication in PVM, by default, goes through pvmds. They act as message routers. Nevertheless this routing has a negative performance impact. To allow faster communication PVM offers a direct communication mode to tasks allowing them to communicate point to point with no intervention of pvmds. This mode is called Direct Routing (see fig. 2). In this mode a task send messages to another through a TCP link. As noted before, by default, a task routes messages to its pvmd, but if direct routing is enabled when a message is passed a direct route is created if one doesn't already exist. The route may be granted or refused by the destination task, or fail, if the task doesn't exist.

2.2 Porting PVM to New Architectures

Several versions of PVM allow the use of fast communication hardware, namely SCI [8] and Myrinet [10]. All these approaches are based on extensive modifications of pvmd. The most difficult part in these modifications is related to the coexistence of:

TCP and UDP sockets used to communicate with other pvmds and local or remote tasks. In the case of TCP a stream communication service is provided. Also, the select primitive allows a non-blocking reception on multiple communication channels.

non-standard communication primitives highly dependent of the hardware used. In most cases, the communication service is based on unreliable datagrams. If a select-like primitive exists, it is not integrated with the one used with sockets.

As the daemon and the PVM library cannot block, they rely heavily on the use of the select primitive over a set of sockets. Although several implementations of sockets over a fast hardware exist [6, 7], these emulations are not perfect.

Other approach used is to restrict the use of the fast communication hardware to the PVM tasks. Standard Internet communication protocols are used by the pvm_{ds}. In this approach overall application performance is improved by optimizing the application communication path.

The concept of direct route described above, can be used to allow tasks to direct access fast communication hardware, still using the traditional PVM primitives. A new option is added to the *pvmsetopt* primitive specifying that the fast communication hardware is used in a point-to-point communication. This is one of the approaches used in [8, 9] and it's our approach too.

2.3 VIA and Message Passing Libraries

The VIA API is very different from the socket API. VIA API is low-level and has none of the high-level features of sockets. VIA specification provides only a minimal set of primitives mainly for user-level data transfer of a single message. Those primitives lack many high-level features such as a stream transport service. For a more detailed discussion of the difficulties of using VIA to support a standard message-passing library please refer to the description of the port of MPI to VIA done at Parma University [11].

To minimize the modifications to the PVM library in the part of direct routing, we build a library that offers a set of socket-like primitives over VIA hardware.

3 Lite Stream Library

The intermediate layer between PVM and VIA that was developed is called Lite Stream Library (LSL).

3.1 Internals

Our LSL library implements a stop-and-wait algorithm. Each message send implies a descriptor. If no descriptor has been posted in the receiver side the incoming message will be lost. The sender only sends the message when he knows that a descriptor is ready to receive it. A two-way handshaking approach method is adopted.

Multi-threading its not used because thread synchronization has a negative impact on latency, so the single threaded application handles descriptors and associated messages when the communication primitive functions are called. Each descriptor has a buffer of 16KB. In the current version, LSL only supports 32 simultaneous connections and only one task per host can use the VIA hardware.

3.2 VIA Implementations on Linux

M-VIA (Modular VIA) was developed by NERSC and aims to provide a modular implementation that can be used for various types of NICs, including Fast

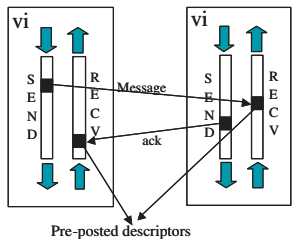


Fig. 3. LSL internal two handshaking protocol

Ethernet and Gigabit Ethernet adapters. For legacy Fast Ethernet cards, M-VIA emulates the VIA specification by software in an intermediate driver layered on top of the standard network driver. This is the case in the hardware used in this work: Fast Ethernet Intel Express Pro NIC.

Table 1 below shows critical values. The bandwidth numbers are obtained with the longest message size (approx. 32 K bytes) and the latency numbers are obtained with the shortest message size.

TCP/IP has a very good performance in very long messages, but in small messages the performance falls down.

3.3 Performance Results

Figure 4 shows LSL performance. We can observe that for short messages our library is better than TCP/IP. The performance difference from M-VIA it’s due to an extra data copy operation between application memory space and library space.

For longer messages LSL is worst than TCP/IP. Registering the application memory helps to solve this problem. However, the penalty for registering the memory region is only compensated when sending a large chunk of data

This library is still in development and we hope to get much better performance.

3.4 Modifying PVM

The modifications to the PVM implementation must not subvert its main characteristic that is the capability to run in heterogeneous environments. The direct

Table 1. Critical values for M-VIA performance versus TCP/IP

| | <i>Bandwidth(MB/s)</i> | <i>Latency(us)</i> |
|--------|------------------------|--------------------|
| TCP/IP | 11,47 | 60 |
| MVIA | 11,71 | 41 |

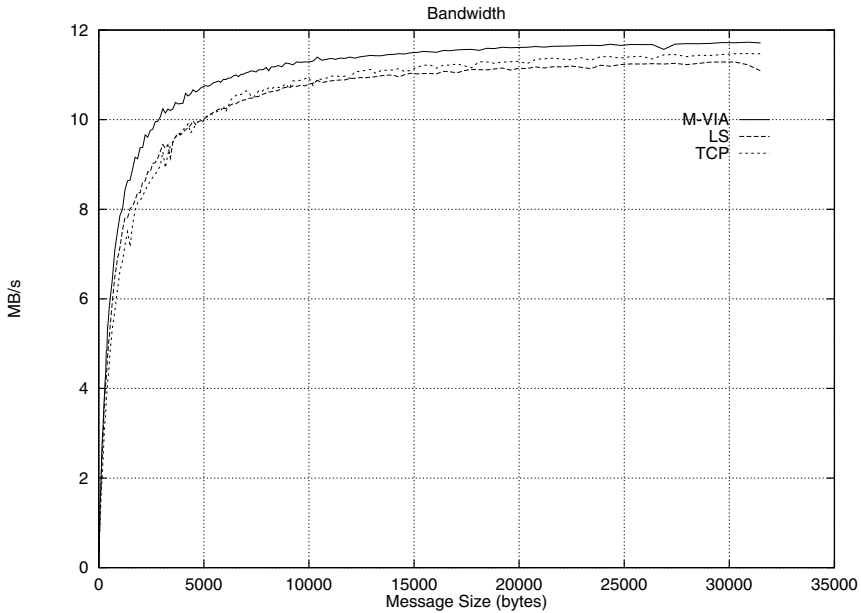


Fig. 4. LSL bandwidth

route improvements over VIA are added to PVM but standard direct route over TCP/IP is kept. This functionality is implemented entirely in the PVM library (libpvm.a). To use direct route over VIA one should specify the value `PvmRouteDirectVIA` to the PVM function `pvm_setopt`. When a task attempts to send a message to other task, the internal PVM library function `mroute()` tries to create a LSL link with the destination task, in case that it does not exist. The destination task must be remote. This implementation is done adding new control messages like `TC_CONREQVIA`, `TC_CONACKVIA` that have the same use and similar meaning of the existing `TC_CONREQ`, `TC_CONACK` control messages.

4 Conclusions and Future Work

In this paper we present a solution to improve PVM over a lightweight library. The results obtained confirm the advantage of the use of lightweight protocols. M-VIA is a solid implementation of VIA, but in the network hardware used a software emulation is made and that has some impact in performance. However, this PVM implementation can take advantage of more efficient VIA-compliant hardware.

Previous work [8,9] has shown that the choice of improving only direct route is enough to increase the performance of PVM applications. The implementation of LSL library to facilitate the integration between VIA and the PVM appears

to be essential in all the work already done. It is important to note, however, that the LSL library is a product by itself, being able to be used directly by the applications. The experimental results show that LSL library has better performance than TCP/IP for small messages.

We are working will be the improvement of LSL library performance and functionality. We can obtain better performance for small messages using a better flow control protocol, like go-back-N, and a more efficient memory register management.

For larger messages, the performance enhancement work is centered around the facilities for memory region registering.

References

- [1] Eicken, T. von , Basu, A., Buch, V., Vogels., W.: UNet: A UserLevel Network Interface for Parallel and Distributed Computing. In Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95), Copper Mountain, Colorado, December 1995 341
- [2] Cameron, D., Regnier, G.: The Virtual Interface Architecture, Intel Press, ISBN 0-9712887-0-4, USA, 2002 342
- [3] Ciaccio, G.: A Communication System for Efficient Parallel Processing on Clusters of Personal Computers, Università di Genova, Genova, Italy, 1999 341
- [4] A. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R. J., Sunderam, V.: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing, MIT, 1994
- [5] Mancheck, R. J.: Design and Implementation of PVM Version 3, MSc Thesis. University of Tennessee, Knoxville USA, 1994 343
- [6] Rodrigues, S.: Building a Better Byte Stream, MSc Thesis, University of California, Berkeley, 1996 344
- [7] Kim, J., Kangho Kim, K., Jung, S.: Building a High Performance Communication Layer Over Virtual Interface Architecture on Linux Clusters, Computer System Research Department, Electronics and Telecommunications Research Institute (ETRI), KOREA, 2001 344
- [8] Fischer, M., Simon, J.: Embedding SCI into PVM, Paderborn Center for Parallel Computing, 1997 344, 345
- [9] Zhou, H., Geist, A.: Faster Message Passing in PVM, Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, 1995 345
- [10] Hristov, D. : Parallel Virtual Machine on Myrinet, Technical Report, ACAPS Laboratory, Mc Gill University, 1996 344
- [11] Bertozzi, M., Panella, M., Reggiani, M.: Design of a VIA based communication protocol for LAM/MPI Suite, Proceedings of Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, 2001 345

LICO: A Multi-platform Channel-Based Communication Library

Moreno Coli¹, Paolo Palazzari², and Rodolfo Rughi¹

¹ University "La Sapienza" - Electronic Engineering Department
Via Eudossiana, 18 - 00184 Rome

² ENEA - HPCN Project - C.R.Casaccia
Via Anguillarese, 301, S.P. 100 - 00060 S. Maria di Galeria (Rome)
palazzari@casaccia.enea.it

Abstract. This paper presents the Light Communication (LiCo) library, designed to implement communications primitives based on permanent, point-to-point, bi-directional channels. LiCo, using the standard TCP layer, allows the interoperability of different computing systems. LiCo library has been designed to be as simple as possible, hiding to the programmers all the unnecessary details connected to initializations and HW heterogeneity management. We report results in the ping/pong test performed between different systems connected through different communication media and we compare, on the same ping/pong, LiCo performances with the ones achieved by PVM and by straight socket communications.

1 Introduction

Different computing architectures may be connected through standard communication infrastructures to allow the execution of distributed computations. Distributed computing systems can be used, for instance, in the case of loosely coupled heterogeneous applications [1-2].

To date, several powerful environments exist to allow the communication among different tasks: among the others, we recall the Nexus Library [3-4] developed within the Globus project [5-6], the ACE Library [7], MPI [8-9], PVM [10], P4 [11], Linda [12]. Many of these systems either are services or require quite large SW installation. Moreover, due to the powerful constructs implemented, they may require long training times. A further aspect that could become a serious drawback is that, as in the case of the Nexus library, the communication environment is characterized by its own security policy, which could be conflicting with the security policy of the hosting computing center.

Due to the previous characteristics of communication environments, we developed the LiCo (Light Communication) library. According to the classical CSP computing model [13], LiCo communications are based on permanent, bi-directional, point-to-

point channels, allowing a simple and intuitive programming style. The main characteristic of LiCo are the following

- it is based on the TCP communication layer (through the standard sockets),
- it is implemented through processes executed in user-space,
- it minimizes the memory copies thanks to the channel concept,
- it defines a very restricted set of communication primitives which are configured by means of their parameters,
- it uses an external process which implements the dynamic resolution of the channel addresses.

2 LiCo Channels

As stated in the introduction, LiCo communications are based on permanent, bi-directional, point-to-point channels.

A LiCo channel is a socket connection which is settled at the first attempt to use the channel and is represented by the pair *<IP address:port number>*, where *IP address* is the address of the processor where the process is executed and *port number* is the port allotted to the channel. It is up to the library, once checked that the channel is not yet initialized, to correctly set up the socket connection. Once the channel has been initialized, it remains alive till the end of the process or the explicit call to the `LICO_Close()` function. After the initialization, each further communication is translated into a direct read/write from/to the socket. This solution is aimed at avoiding the explicit initialization steps normally required by other communication libraries (like PVM or MPI), thus reducing the number of primitive functions to be inserted into the library.

The potential use of more than one channel between two processes allows the exploitation of the communication parallelism within the processes, being each channel associated to its communication thread. Furthermore, at the cost of using several TCP ports between two communicating processors, the number of memory copies needed for the received message is reduced: in fact the incoming message is directly copied into its destination area without any need of further analysis to decide its right storing address. On the contrary, the using of only one channel between two any communicating processors, while minimizing the number of TCP ports, will cause the need to analyze the header of the incoming message in order to decide its correct final memory address, thus implying an additional memory copy.

LiCo channels are accessed only through the following send and receive functions, defined as

```
LICO_MsgSend(char *chan_id, tp_lico_msg *msg, tp_lico_cmd lico_cmd)
LICO_MsgRecv(char *chan_id, tp_lico_msg *msg, tp_lico_cmd lico_cmd)
```

which are used to send (receive) the message `msg` in (from) the channel identified by `chan_id`. All messages are constituted by an integer, representing the length in bytes of the message, followed by the body of the message. The parameter `lico_cmd` may assume the value 'WAIT' or 'NOWAIT' and is used to specify whether a communication is blocking or not.

Four queues (S_1, S_2, R_1, R_2) are associated to each communication channel in order to implement the not-blocking communications: two $-S_1, S_2-$ are used for the send and two $-R_1, R_2-$ for the receive operations. The queues contain the pointers to the messages. In order to keep unaltered the right message ordering, the blocking operations are performed only after that the queues have been emptied.

The two queues, both at the sending and at the receiving side, are used to simply decouple the user operations from the LiCo operations, thus avoiding mutual accesses to a single queue: while the queue S_1 (R_1) is used to queue the messages, the other is used to effectively transmit the previously queued messages. After each transmission the queues are swapped.

3 Address Resolution: The RT-PAT Process

LiCo library uses permanent, point-to-point, bi-directional channels. In order to implement such a communication modality, we must be able to reserve an arbitrarily large number of channels (i.e. TCP ports) to each process – obviously not exceeding the maximum number of ports allowed, namely 64512. The ports are system resources, so a port cannot be statically linked to a channel because it could be reserved by another process; when creating a channel its port number must be dynamically chosen among the not reserved ports. When a process tries to execute its very first communication operation, it starts a loop to reserve as many ports as they are needed by the process (this number is statically determined). Once the ports are reserved, they are put in correspondence with their ‘virtual’ address, i.e. the `chan_id` string. Thus a translation table is implemented to map each channel of a process into a port (Table 1). The translation table has three heading lines specifying the IP address, the machine word size and the little/big endian information for the process. The information on the word size and the endian type of the system are needed for heterogeneous nature of the distributed systems and are used only to process the header of the message; further decoding operations on the message are left to the user.

Table 1. Virtual-physical port number translation table

| IP address | | |
|-------------------|---|-----------------|
| machine word size | | |
| endian type | | |
| chan id l | → | port number l |
| ... | → | ... |
| chan id n | → | port number n |

Before a process can communicate with another process, the channel has to be initialized, i.e. the socket connection corresponding to the channel has to be set up. In order to do this, the process which is trying to start the channel connection must know the address of the remote socket, that is the IP address and the port reserved by the process on the other side of the channel. We devised a global service, called Run Time – Port Address Translator (RT-PAT), which is active on a known socket and is in charge to keep track of the channel translation tables of all the processes; we

underline that the socket (*IP address:port*) of RT-PAT is the only resource that must be known (and reserved) *a priori*.

The RT-PAT process is polled by the processes that, to set up a communication, need the IP address and the port number of their remote sockets. Figure 1 highlights the communication steps necessary to allow processes A and B to communicate:

- as soon as process A and B are started, before performing their first communication, they both send their translation table to the RT-PAT process (A arrows)
- when attempting to communicate for the first time (channel initialization – D arrows), A and B processes poll RT-PAT to know the IP address and the port number of the other side of the channel; RT-PAT replies with the requested information (B arrows);
- the socket connection between processes A and B is set up and all the subsequent communications $A \leftrightarrow B$ are performed as direct socket communications (C arrows)

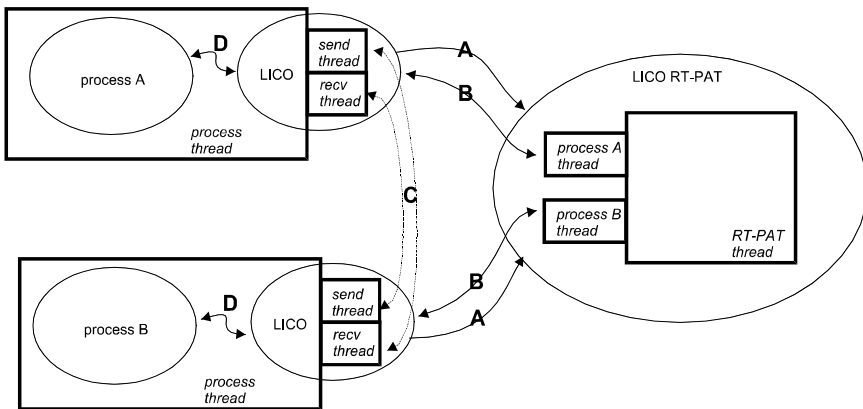


Fig. 1. A: each process sends to RT-PAT its allocation map; B: the processes ask RT-PAT for the allocation maps of other processes and receive them; C: the processes communicate directly without using RT-PAT; D: send and receive library calls

4 Results

One of the main goals writing LiCo was to keep the programming style as simple as possible, hiding to the programmer all the unnecessary details and defining few primitive functions powerful enough to represent a large class of parallel programs.

In order to illustrate the programming style with LiCo, we report the C programs used in the ping pong tests. The two processes, connected through the channel labeled '1', send and receive 100 times a message of 1000 bytes. As the communication functions are invoked with the command parameter 'WAIT', communications have a blocking behavior.

The `LICO_Create_msg(int msg_size)` is a function which creates a message structure allocating `msg_size` bytes. In the PONG program it is called the `LICO_Create_msg(0)` because it is up to the `LICO_MsgRecv` to allocate the memory for the incoming message.

```

/* LICO PING */
#include "lico.h"
int main()
{tp_lico_msg* msg;
 int i;
 msg = LICO_Create_msg(1000);
 for (i=0;i<100;i++) {
  LICO_MsgSend('1',msg,WAIT);
  LICO_MsgRecv('1',msg,WAIT)};
 free(msg);
 LICO_Close();
}

/* LICO PONG */
#include "lico.h"
int main()
{tp_lico_msg* msg;
 int i;
 msg = LICO_Create_msg(0);
 for (i=0;i<100;i++) {
  LICO_MsgRecv('1',msg,WAIT);
  LICO_MsgSend('1',msg,WAIT)};
 free(msg);
 LICO_Close();
}

```

It is worth to be underlined that LiCo functionalities may be accessed without any initialization step. In order to be run, previous programs need the RT-PAT process to be active.

To test LiCo library in heterogeneous environment we ran the ping/pong test between different systems, some within the same LAN constituted by several sub-nets (10/100 Mb/s) and other belonging to different LANs connected through shared channels with different bandwidths (ranging from 2Mb/s up to 8 Mb/s). The length of messages varies from 2 bytes up to 32 Kbytes.

We refer to the following three test-bed configurations:

C1: two alpha systems (processor EV6.7@667 MHz, OS Linux 2.4.0, gcc compiler invoked with the `-O3` option) connected through a fast-ethernet hub;

C2: one alpha system (processor EV6.7@667 MHz, OS Linux 2.4.0) and one node of an IBM SP2 parallel system (processor power3@200 MHz, OS AIX 4, C compiler for AIX – version 4.4 – invoked with the `-O3` option) belonging to different LANs connected through a 4Mb/s channel;

C3: one SGI system (processor R12000, OS Irix64 6.5, MIPSpro C compiler – versione 7.3.1.1m – invoked with the `-mips4 -O3` options) and one Sun system (Sun UltraSparcII@450 MHz, OS Solaris 2.7, Sun WorkShopc C compiler – version 5.0 - invoked with the `-O3` option) belonging to different sub-nets within the same LAN.

The tests were repeated 1000 times in order to diminish the effect of traffic load on the measured times. From the measured data we computed the communication start-up time and the maximal bandwidth, reported in table 2.

Table 2. Minumum start-up time and maximal bandwidth as measured in ping/pong tests

| Testbed configuration | Start-up time (msec) | Bandwidth (Mbit/sec) |
|-----------------------|----------------------|----------------------|
| C1 | 0.06 | 53.5 |
| C2 | 26.5 | 2.39 |
| C3 | 0.48 | 7.56 |

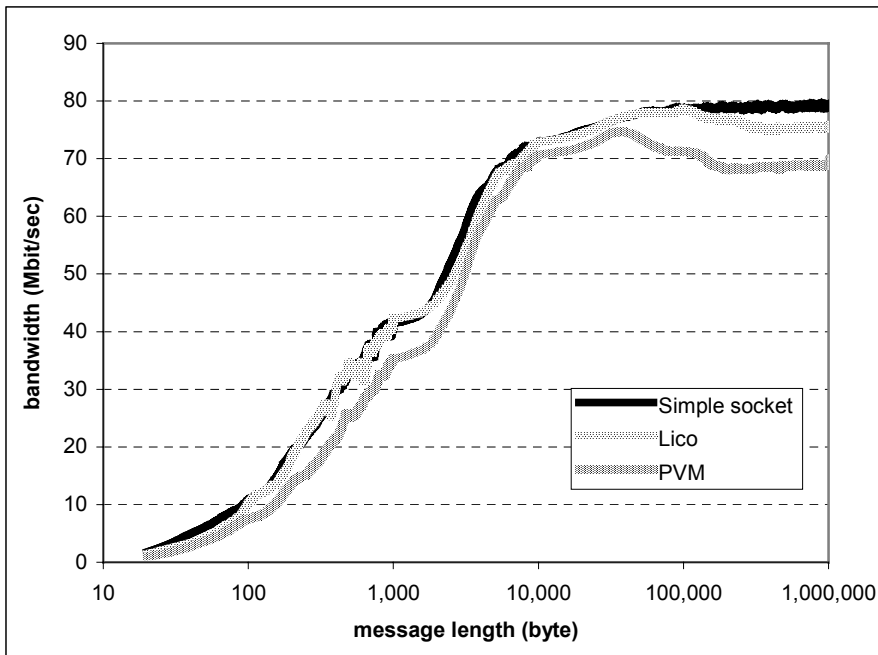


Fig. 2. Comparison among LiCo, PVM and simple socket communication

The successive tests were devoted to obtain a comparative measure among PVM (we used the 3.4 version with the *direct_route* and *pvm_data_raw* modality), LiCo and the direct socket connection (which is the absolute minimum which could be achieved - without any system optimization - by TCP based libraries). The measured data refer to two PC Athlon@1300 MHz connected through a fast ethernet hub and completely dedicated to run the test. Both the PC were equipped with the Linux Redhat 7.2 OS.

The analysis of Figure 2 evidences a substantial equivalence between socket and LiCo for messages ranging from few bytes up to 100 KBytes, while socket communications are more efficient for longer messages, being characterized by nearly 80 MB/s of bandwidth against the 76 MB/s of the LiCo. The comparison between LiCo and PVM clearly evidences the superiority of LiCo which, especially for long messages, takes advantage from the less number of memory copies for the incoming messages and gives asymptotic bandwidth figures of 76 MB/s against the PVM bandwidth which is close to 69 MB/s.

5 Conclusions

The need for a simple communication library working exclusively as user process motivated the design and implementation of the Light Communication (LiCo) library.

The LiCo library has been presented in this work, illustrating its structure along with its communication functions.

LiCo library allows process-to-process communications by means of permanent, point-to-point, bi-directional channels. A dedicate process exists (the RT-PAT process allocated on a system connected to the network) which dynamically maps LiCo channels onto physical TCP ports.

Special efforts has been spent to make the LiCo programming style as simple as possible. The ping pong test has been presented as example of the LiCo library use. The roundtrip time for different test-bed configurations was measured using different communication primitives based on the basic socket structure, on the PVM functionalities and on the LiCo library.

The test-bed infrastructure refers to three different configurations: two systems within the same LAN and the same sub-nets, two systems within the same LAN but in different sub-nets, two systems in two different LANs.

The results show a very satisfying behavior: up to 75% of the maximal network bandwidth is used and the start-up time ranges from 0.06 ms up to 27 ms, depending on the network connection.

Furthermore, the comparison between LiCo and PVM in the ping pong case clearly demonstrates the better performances attainable with the LiCo library.

References

- [1] Palazzari P. et al, Heterogeneity as Key Feature of High Performance Computing: the PQE1 Prototype, *Proceedings of the 13th Heterogeneous Workshop*, Cancun, Mexico, May 2000.
- [2] See the url <http://setiathome.ssl.berkeley.edu/>.
- [3] Foster I., Kesselman C., Tuecke C., The Nexus Approach to Integrating Multithreading and Communication, *Journal of Parallel and Distributed Computing*, 37:70--82, 1996.
- [4] Foster I., Kesselman C., Tuecke C., The Nexus Task-Parallel Runtime System, *Proceedings of the 1st Int'l Workshop on Parallel Processing*, 1994.
- [5] Foster I., Kesselman C., Globus: A Metacomputing Infrastructure Toolkit, *Intl J. Supercomputer Applications*, 11(2):115-128, 1997.
- [6] Czajkowski K., Fitzgerald S., Foster I., Kesselman C., Grid Information Services for Distributed Resource Sharing, *Procedings of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, IEEE Press, August 2001.
- [7] Schmidt Douglas C., The ADAPTIVE Communication Environment, *Proceedings of the Sun User Group Conference*, San Jose, California, December 1993.
- [8] Dongarra J.J., Walker D. W., MPI: A Standard Message Passing Interface, *Supercomputer*, Vol. 12, No. 1, pages 56-68, January 1996.
- [9] Snir M., Otto S.W., Huss-Lederman S., Walker D. W., Dongarra J. J., *MPI: The Complete Reference*, <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>.

- [10] Sunderam V.S., PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience*, Vol. 2(4), December, 1990.
- [11] Butler R., and Lusk E., Monitors, messages, and clusters: The p4 parallel programming system, *Parallel Computing*, Vol. 20, 1994.
- [12] Carriero N., Linda in Heterogeneous Computing Environments, *International Parallel Processing Systems IPPS*, 1992.
- [13] C.A.R. Hoare, Communicating Sequential Processes,. *Prentice Hall International Series in Computer Science*, 1985.

Notes on Nondeterminism in Message Passing Programs

Dieter Kranzlmüller¹ and Martin Schulz²

¹ GUP, Joh. Kepler University Linz
Altenbergerstr. 69, A-4040 Linz, Austria/Europe
kranzlmueeller@gup.uni-linz.ac.at
<http://www.gup.uni-linz.ac.at/~dk>

² Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik der Technischen Universität München
D-80290 Munich, Germany
schulzm@in.tum.de
<http://wwwbode.in.tum.de/~schulzm/>

Abstract. Nondeterministic program behavior can lead to different results in subsequent program runs based on the same input data. This kind of problem can be seen in any program, but is even magnified in a parallel execution context due to the existence of several independent but communicating tasks. Even though this kind of nondeterminism is commonplace and in many cases even useful for the implementation of applications, it often leads to sporadically occurring errors. These bugs are difficult to reproduce and represent a heavy challenge during testing and debugging. The biggest problem, however, may be the unawareness of users about the existence of nondeterministic choices and their consequences. In order to raise the awareness and to provoke discussions about this serious problem, this paper provides an exemplary overview of nondeterministic behavior in message passing programs. With simple examples, it is demonstrated how nondeterminism can vigorously affect the behavior and the final results of software and how the behavior can change between different architectures.

1 Introduction

With networks of workstations getting an appealing vehicle for parallel processing and corresponding parallel software tools hiding the system's complexity, High Performance Computing (HPC) is available to a rapidly growing user community [1]. However, serious pitfalls and problems exist which often cause troubles in the use of HPC, even for the experienced user.

One of the most crucial examples of such a pitfall is nondeterministic program behavior. Whenever situations exist, where a program's flow depends on the outcome of a nondeterministic statement, the users may be confronted by a set of problems affecting a program's operation. In this case, a program may produce different results, although the same input data is provided and no modifications

to the source code or the binary have been applied [15]. Instead, the results may depend on environmental factors, which cannot be controlled by the user (e.g. variations in processing speed, scheduling decisions, or contention in the network) [12, 7]. The observation of different program results is often sufficient to initiate program analysis activities. More difficult situations may occur when nondeterminism remains unnoticed during testing and debugging. In this case, users may rely on their application’s fault-free operation, while the program still holds the possibility for serious bugs. The consequences may only be noticed at a much later point of time, when corrections are much more difficult (or even impossible) to apply. In addition, nondeterminism may affect portability, since different machines with their different execution environments induce different behavior, even though the same code or even binary is used.

While such nondeterministic behavior is already known in sequential programs, e.g. due to timing mechanisms, it is potentially more severe in parallel environments with several independent tasks operating in loose cooperation. This paper focuses on message passing programs. In this context, the term “race condition” characterizes the situation, where a program’s flow depends on the order of process interaction statements [4]. In contrast to programs relying on the shared memory programming paradigm, the occurrence of race conditions in message-passing codes is explicit, since they may only occur at a limited set of statements and hence can be more easily characterized [14].

This paper emphasizes on this situation and shows how easily and at the same time how significantly nondeterminism can affect the outcome of a code. The experiments for this study have been carried out on a large number of machines showing the impact of different execution environments on program behavior. To our knowledge, no comparable experimental evaluation has been performed in the past. Even though this paper focuses on message passing codes, the consequences of nondeterministic behavior are comparable in any paradigm and hence the observations of this paper may be directly transferred onto any other (parallel) programming paradigm.

The paper’s structure is as follows. Section 2 justifies the reasons for using nondeterminism in applications, and describes the possible sources of nondeterminism in message passing programs in more detail. The main focus of this paper are the examples in Section 3, which describe different nondeterministic situations in message-passing programs and their impact on different architectures. The paper is then concluded by looking on possible consequences and solutions in Section 4.

2 Nondeterministic Behavior in Message-Passing Programs

Although many different programming paradigms for parallel and distributed computing have been proposed, message-passing is still the most widely used paradigm for HPC. While its drawbacks are well-known in the community, no better suited approach has been defined for achieving efficiency and portability

across the huge set of available parallel architectures. A large portion of this success can certainly be attributed to the standardization initiatives in the area of messaging APIs, which have lead to mainly two dominant libraries, namely the Message-Passing Interface (MPI) [13] and the (by now less dominantly) Parallel Virtual Machine (PVM) [3]. Both approaches share the characteristic of separate address spaces for each process, so that any data exchange or synchronization must be performed via explicit communication operations.

In message-passing programs, nondeterministic behavior may be observed at receive operations and system calls [8]. The latter describes operations, which explicitly interact with the operating system and are therefore highly dependent on the computing environment. A well-known example is the random number generator, which delivers an arbitrary value within a certain range of numbers. Other examples are the `gettimeofday()` function call, operating system interrupts and signals [18].

This paper focuses on the message passing aspects of nondeterministic behavior, in which the receive operations are of major importance. In this case, nondeterminism may occur at wild card receive operations, nonblocking receives, or global operations, where the arrival order of messages may influence the program's behavior in an undefined way¹. A call to a receive operation with a wild card (e.g. `MPI_ANY_SOURCE`) as the source identifier allows a message from any process to be accepted. Consequently, the execution of a particular process after the receive operation may depend on the arrival order of messages at potential race conditions. The corresponding messages are so-called “racing messages”. An example for this is given in Figure 1. The graphs symbolize the execution of three processes (P,Q,R) along the horizontal time axis. Communication is denoted by dashed arrows pointing from send points at the message source to the corresponding receives in the destination processes. Assuming process P issues a receive operation (a) with a wild card parameter, while both other processes issue a corresponding send. In this case, either of the two execution scenarios of Figure 1 may be observed. In the left execution, message (b) from process Q is accepted, while in the right execution, message (c) from process R arrives first on process P. Depending on the messages' arrival order, the execution of the three processes is different.

Such a nondeterministic situation can lead to one of the following consequences:

- Nondeterministic behavior may be fully compensated by the application, e.g. by fully independent message handling, and may not affect the execution at all.
- The nondeterministic behavior may remain unnoticed, if only one of both execution orders occurs during testing; the second one remains completely untested.
- Errors may occur only sporadically, if one of both execution orders has a higher probability, while the other contains a bug.

¹ Due to the limited space, this paper focuses only on blocking receive operations using wild cards; most issues are, however, comparable for the other operations [8].

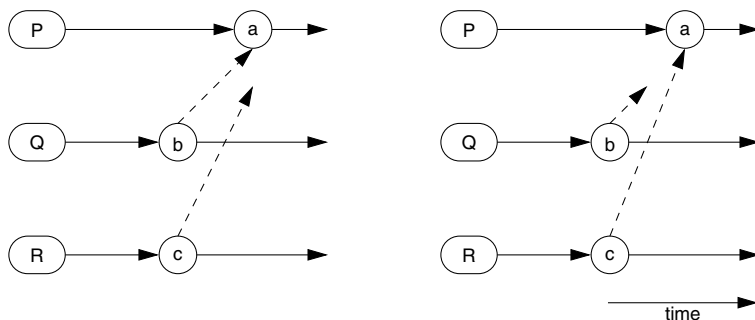


Fig. 1. Nondeterministic behavior at wild card receives

- A particular program run cannot be re-executed at will, because any re-execution may deliver any of the two possibilities with similar probability.

The last three scenarios lead to an unpredictable execution of the corresponding code and may impact both the correctness and the stability of the program. It is important to notice though, that in case of potential misbehavior, the character of nondeterminism, as lined out above, only partly depends on the application code itself. Probably even more important is the respective execution environment in the form of the underlying parallel architecture and the operating system properties. This may lead to the consequence that the program behavior may change unexpectedly when moving to other platforms and hence severely impair the portability of the code.

With all these problems raised by nondeterministic program behavior and wild card receives in particular, it remains to be discussed why nondeterminism is not “prohibited” or purposely avoided during the design of the application in the first place. At this point it is important to notice, that nondeterministic behavior by itself is not erroneous [8]. In many cases it is even useful and often opens opportunities for performance improvements by providing additional degrees of freedom. For instance, with wild card parameters at receive operations, simple first-in-first-out (FIFO) queues can be installed [15], enabling mechanisms like adaptive load balancing through task redistribution or task stealing. Another important reason is reality itself, which often exhibits nondeterministic characteristics, e.g. Chaos Theory or Monte Carlo methods. To model the same behavior in computer simulations, nondeterministic behavior must be available as a feature for the software developer. Finally, more related to human nature than to technical reasons, it is often convenient to use wild cards instead of computing the origin of a message, especially in case of complicated routing mechanisms.

In many cases, however, programmers simply underestimate the consequences of using wild cards and oversee the potential malicious effects, which can go as far as deadlocks, livelocks, stampede effects, or bystander effects [19].

3 Examples of Nondeterministic Program Behavior

In order to further illustrate the impact of nondeterminism in message passing codes and to discuss the consequences, this section covers a few examples. It is important to note that neither of them are artificially created, but rather were taken from real-world user experiences and hence represent a realistic scenario.

The first example describes a simple program fragment taken from of a larger application, which was developed by students during their practical work. The communication pattern induced by this fragments is shown in Figure 2. Each of the four screenshots shows a space-time diagram with several communicating processes over time in the horizontal axis and their explicit communication behavior [7].

All four diagrams represent the execution of the same program, but at two separate execution instances (top vs. bottom). Each set of diagrams is additionally plotted in two different time scales with physical time in milliseconds being shown in the left diagrams and logical time in the the right ones with distance drawn as multiples of a fixed interval. While the former diagram type is usually applied for performance analysis activities, the latter is extensively used in correctness debugging. In the example of Figure 2, the logical clock display is useful to see, that each process contains six distinguishable events (three sends and three receives), which is not obvious from the real-time diagram due to an overlapped display of some of the send and receive events.

Comparing the execution of the upper and lower diagrams, clear differences in the communication can be observed in both diagram types when examining the first receive operation in P2. While in the first execution, the operation accepts a message from P1, the second sequence leads to an reception of data from P0. This is only possible, if the receive event on process P2 uses a wild card for the origin of a message.

The reason for this difference in the execution behavior of the same program with the same input is determined by two factors: the computation time between the send and the receive event or the communication/transfer time of a message can vary between the two cases. Consider the scenario shown at the bottom example of Figure 2: the computation time between the first send event of process P7 (to process P0) and the second send event of process P0 (to process P2) plus the respective transfer times is less than the transfer time between process P1 and process P2. The reason is most likely some external delay on the node of process P1. Thus, the first message arriving at the first receive event on process P2 is the one from process P0. Further analysis is unfortunately not possible based on this data, as the diagrams only show the correct timing of the actual receive events, not the actual delivery time of a message (which could potentially be much earlier).

The lesson that can be learned from this example is that temporal relations between parallel processes may vigorously influence the behavior of a program in case of nondeterminism. Assuming, that the computation time or the communication time is equally slow on all processors leading to an even distribution of the communication points, this program may always deliver the results of the

top diagrams, while a faster execution of one or more processes may lead to cross-overs in the message delivery. This effect was experienced during a lecture on parallel computing, where students implemented their home-work on a slower parallel machine, while the instructor evaluated their work on a faster machine. Whereas the students always observed only correct program behavior, the instructor recognized only the erroneous results of the code.

While the above example shows how arbitrary message sequences can be provoked, the second set of experiments analyzes the frequency of particular message sequences in a clearly nondeterministic code. For this purpose, the following code is used, in which a set of slaves waits for a random time and then sends a message to a master process, while the master simply records the order of their arrival by printing a sequence based on the respective source node IDs.

```
if(my_rank==0) {                                     // Master node
    for(i=1;i<number_of_nodes;i++) {                 // Receive from all slaves
        MPI_Recv(&msg,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
        printf("%d",msg);                             // Print sending rank
    }
    printf("\n");
}
else {                                                // Slave node
    random_wait();                                     // Wait random time
    MPI_Send(&my_rank,1,MPI_INT,0,0,MPI_COMM_WORLD); // Send to master
}
```

With such a code, most users are aware of the different possibilities that the program may return (at the `printf` statement); it is even expected that every program run delivers a different result due to the nondeterministic nature. Since the MPI definition does not impose any priorities between processes, users may assume that each possible message order shares the same occurrence probability. Thus, in theory, if enough executions are initiated, at one point all possibilities should have occurred. Furthermore, since an intrinsic characteristic of MPI is portability, the same logical behavior is expected on any kind of parallel computing architecture.

In reality, though, the situation is quite different as shown by the plots in Figure 3. Each diagram shows 10.000 results observed for a particular architectures, randomly selected from at least 250.000 independent executions on eight nodes. The machines thereby differ in their base architecture and their interconnection technology and range from various cluster configurations to highly parallel MPPs. In all graphs, the execution number is given on the horizontal axis, while the observed results are displayed on the vertical axis.

For the given code fragment $7! = 5040$ different results could occur (from 1234567 to 7654321). Given our expectations above, these results should be equally distributed across the plots. Yet, this is certainly not the case, with each plot emphasizing different areas. It clearly shows that each machine has different

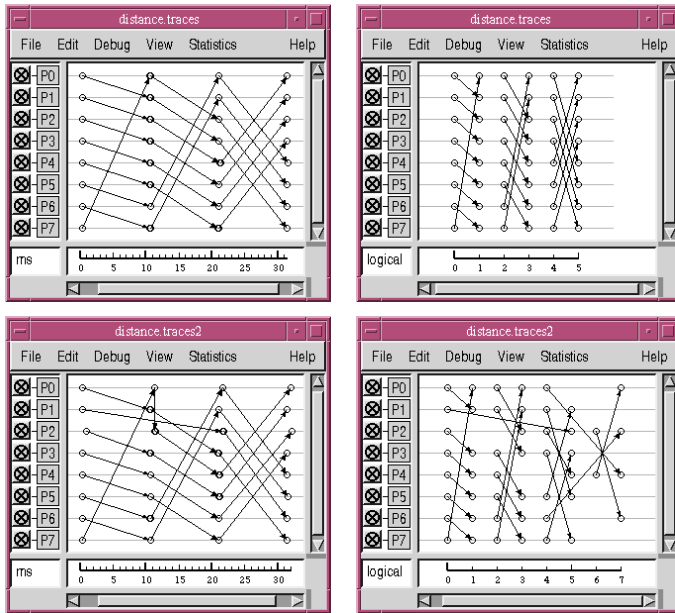


Fig. 2. Nondeterministic behavior due to timing differences of two executions of the same program (top vs. bottom graphs) using either physical (left) or logical time (right)

characteristics with respect to nondeterminism and it also shows that dominant results on one architecture may be of no significance on others.

A good indicator for the degree and character of the nondeterminism found on each of the machines is the observed coverage of all potential execution orders. With just these six machines under observation, this parameter ranges from as low as 1,63% in case of the SGI Origin 3000 to as high as 63,23% in case of the Myrinet cluster. It should be noted, though, that even the latter with the highest coverage by far does not include all possible execution orders. A second interesting criteria, which can be clearly observed, is the dispersion of the results. In case of the SMP cluster interconnected with an SCI ringlet, no single execution order contributes more than 3.3% to the total set of execution orders (which in this case were 1967), while in the Ethernet cluster a single order covers 81% or on the Hitachi system seven orders are responsible for 99,7% of the observed outcome.

A concrete explanation for the observed behavior for each machine, however, is difficult to give, as this may depend on several different factors potentially arising from minor details: the behavior in case of the Ethernet cluster could stem from an asymmetric system setup for node 4. The difference between the two SCI setups may be caused by their different network topologies, SMP configurations,

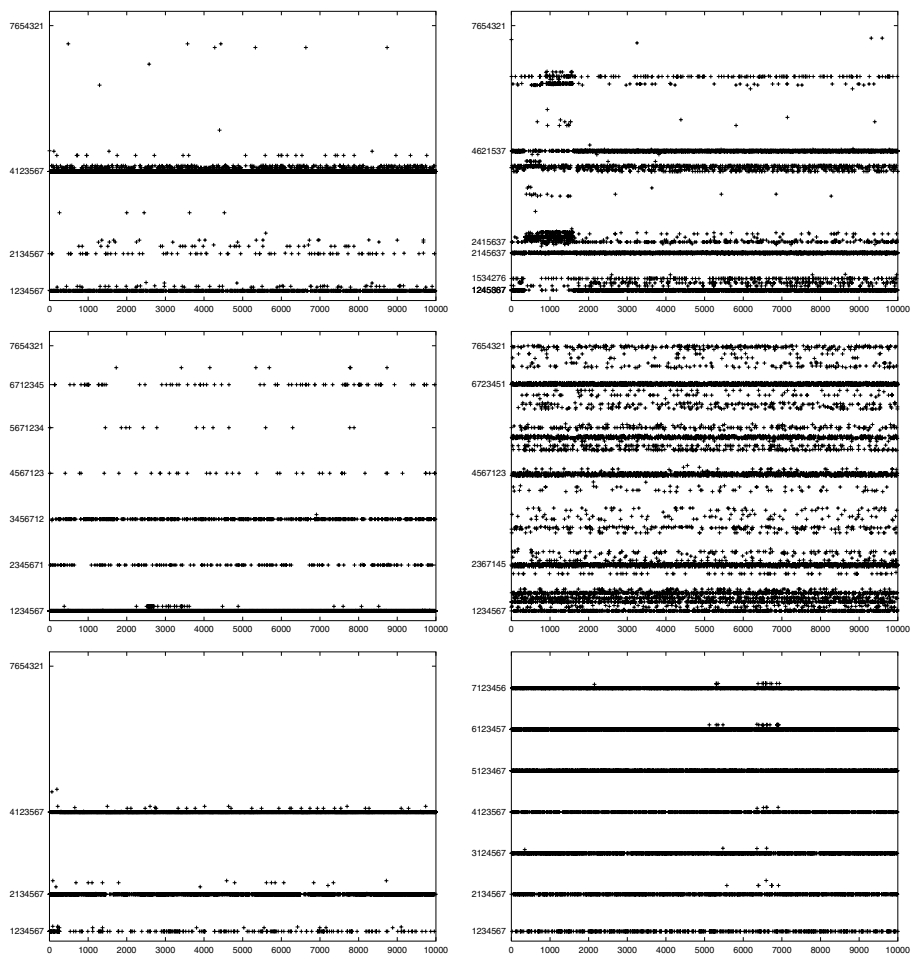


Fig. 3. Nondeterministic behavior of portable programs on different hardware architectures — Switched Ethernet cluster using 4 SMP nodes / LRR-TUM (top left), Myrinet cluster in star topology / ENS Lyon (top right), SCI cluster using a torus / PC2 Paderborn (middle left), SCI cluster using 4 SMP nodes on a ringlet / LRR-TUM (middle right), SGI Origin 3000 / JKU Linz (bottom left), Single Hitachi SR8000 SMP node with 8 CPUs / LRZ München (bottom right)

or Linux scheduler versions. The graph for both the Myrinet and the SGI system seems to somewhat reflect the network topology and the node distances. The behavior of the Hitachi SMP node with its optimal and fully symmetric network is most likely caused by the MPI implementation which seems to check first for messages from nodes with lower IDs. The important conclusion in any case, however, is that nondeterminism clearly can depend on all parameters of the underlying target architecture and is therefore mostly unpredictable, a property which is naturally also inherited by the applications using these architectures.

4 Consequences and Outlook

Nondeterminism can be found in many scenarios. It is especially evident in the context of parallel programming where several loosely related, independent tasks work cooperatively on a common problem and exchange information for both data and control purposes. For example, in message passing programs, wild card receives may lead to unpredictable orders of message arrivals. While useful or necessary in many scenarios, nondeterminism can lead to serious problems when being ignored during application design or testing.

This kind of behavior has been demonstrated using an example in which the message order and hence the program execution is easily affected by a different process or message timing. On the other hand, a second set of experiments has shown that, despite a fully symmetric setup of a nondeterministic execution, only a certain subset of all execution orders can actually be observed on a particular architecture. Additionally, the number of observable executions changes dramatically between different target architectures. This demonstrates how easily undetected nondeterminism can affect the portability of codes, e.g. execution orders containing bugs which do not appear or are very unlikely on one architecture, may be dominant on others leading to severe problems.

Until now, there is no complete solution for all these problems. However, certain approaches address some of the critical issues. The irreproducibility effect is addressed by two kinds of tools, *record and replay* mechanisms [11, 15, 14] and *controlled execution* [2, 20]. The former guarantee equivalent re-execution by observing a program's nondeterministic choices during an initial record phase and using this information during subsequent replay phases. The latter apply rules to define the selection of nondeterministic choices, such that the usage of the same rules will lead to an equivalent execution.

Similarly, the problem of complete testing is addressed by extensions of *record and replay* tools [10] and *controlled execution* tools with corresponding functionality [5]. The former apply event manipulation to modify the behavior observed during the initial record phase, and enforce this behavior during an artificial replay phase. The latter use extended rules for controlled execution, where subsequent program runs apply systematic variations at nondeterministic choices. Examples of using *record and replay* techniques for shared shared memory are described in [17], while an event manipulation is described [6]. This demonstrates the applicability of the same technique for other types of parallel programs.

While all these tools represent standalone testing and debugging tools, the approach described in [9] is integrated into the message passing environment, in this case MPI. The advantage of this approach is that observation and detection of nondeterminism is transparently performed by the message-passing library. In case of re-execution, the message-passing library offers corresponding functionality. This approach represents a suitable extension for analyzing nondeterministic programs, especially for the inexperienced user.

In the long run, such tools could be combined with nondeterminism traces as discussed in the second half of Section 3 in order to test applications with the most likely execution orders found in potential target machines for particular nondeterministic constructs. This might lead to a more predictable port of applications to other architectures and could result in an increased software reliability. Such approaches, however, certainly face a large number of challenges on how to describe and capture the exact nondeterminism characteristics of a target architecture and thereby opens a large area of future research.

Acknowledgments. Several colleagues contributed to this project, either by offering valuable ideas or by giving access to their machines. Some of them are Prof. Dr. Jens Volkert (GUP Linz), Johann Messner (ZID, Univ. Linz), Rade Kutil (Univ. Salzburg), Andreas Schmidt (KONWIHR/LRR, TU-München), and Roland Westrelin (RESAM, Lyon).

References

- [1] R. Buyya (Ed.). *High Performance Cluster Computing - Architectures and Systems*, Vol. 1, and *High Performance Cluster Computing - Programming and Applications*, Vol. 2, Prentice-Hall PTR, New Jersey, USA (1999). 357
- [2] R. H. Carver and K. C. Tai. *Replay and Testing for Concurrent Programs*. IEEE Software, Vol. 8, No. 2. pp. 66-74 (March 1991). 365
- [3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994. 359
- [4] D. P. Helmbold and Ch. E. McDowell. *Race Detection - Ten Years Later*. in: M. L. Simmons, A. H. Hyes, J. S. Brown and D. A. Reed *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 101-126 (1996). 358
- [5] P. Kacsuk. *Systematic Macrostep Debugging of Message Passing Parallel Programs*. Proc. DAPSYS 98, Future Generation Computer Systems, North-Holland, Vol. 16, No. 6, pp. 597-607 (April 2000). 365
- [6] R. Kobler, D. Kranzlmüller, J. Volkert. *Debugging OpenMP Programs Using Event Manipulation*. Proc. WOMPAT 2001, Intl. Workshop on OpenMP Applications and Tools, West Lafayette, Indiana, USA, pp. 81-89 (July 2001). 365
- [7] D. Kranzlmüller, S. Grabner, and J. Volkert. *Debugging with the MAD Environment*. Journal of Parallel Computing, Elsevier, Vol. 23, No. 1-2, pp. 199-217 (April 1997). 358, 361

- [8] D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. Ph.D. thesis, GUP Linz, Johannes Kepler University Linz, Austria (September 2000) <http://www.gup.uni-linz.ac.at/~dk/thesis>. 359, 360
- [9] D. Kranzlmüller, Ch. Schaubschläger, and J. Volkert. *An Integrated Record&Replay Mechanism for Nondeterministic Message Passing Programs*. Proc. EuroPVMMPI 2001, Springer, LNCS, Vol. 2131, pp. 192-200 (2001). 366
- [10] D. Kranzlmüller and J. Volkert. *NOPE: A Nondeterministic Program Evaluator*. Proc. of ACPC '99, 4th Intl. ACPC Conference, Springer, LNCS, Vol. 1557, Salzburg, pp. 490-499 (Feb. 1999). 365
- [11] T. J. LeBlanc and J.M Mellor-Crummey. *Debugging Parallel Programs with Instant Replay*. IEEE Transactions on Computers, Vol. C-36, No. 4, pp. 471-481 (April 1987). 365
- [12] Ch.E. McDowell and D.P. Helmbold. *Debugging Concurrent Programs*. ACM Computing Surveys, Vol. 21, No. 4, pp. 593-622 (December 1989). 358
- [13] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. <http://www.mpi-forum.org/>. 359
- [14] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. *Debugging Race Conditions in Message-Passing Programs*. Proc. SPDT 96, ACM SIGMETRICS Symposium on Parallel and Distribution Tools, Philadelphia, PA, USA pp. 31-50 (May 1996). 358, 365
- [15] R. H. B. Netzer and B. P. Miller. *Optimal Tracing and Replay for Debugging Message-Passing Parallel Program*. Proc. Supercomputing 92, Minneapolis, MN, USA, pp. 502-511 (November 1992). 358, 360, 365
- [16] M. Oberhuber. *Elimination of Nondeterminacy for Testing and Debugging Parallel Programs*. Proc. AADeBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging, Saint Malo, France, pp. 315-316 (May 1995).
- [17] M. Ronsse, K. DeBosschere. *RecPlay: A Fully Integrated Pratical Record/Replay System*. ACM Transactions on Computer Systems, Vol. 17, No. 2, pp. 133-152 (May 1999). 365
- [18] M. A. Ronsse, K. De Bosschere, and J. Chassin de Kergommeaux. *Execution Replay and Debugging*. Proc. AADeBUG 2000, 4th Intl. Workshop on Automated Debugging, Munich, Germany, pp. 5-18 (August 2000). 359
- [19] D. F. Snelling and G.-R. Hoffmann. *A Comparative Study of Libraries for Parallel Processing*. Proc. Intl. Conf. on Vector and Parallel Processors, Computational Science III, Parallel Computing, Vol. 8 (1-3), pp. 255-266 (1988). 360
- [20] K. C. Tai, R. H. Carver, and E. E. Obaid. *Debugging Concurrent Ada Programs by Deterministic Execution*. IEEE Transactions on Software Engineering, Vol. 17, No. 1, pp. 45-63 (January 1991). 365

Web Remote Services Oriented Architecture for Cluster Management*

Josep Jorba, Rafael Bustos, Àngel Casquero, Tomàs Margalef, and Emilio Luque

Computer Architecture and Operating Systems Group (CAOS)
Escola Tecnica Superior de Enginyeria
Universitat Autònoma de Bellaterra (UAB)
08193 Bellaterra, Spain
{josep.jorba,tomas.margalef,emilio.luque}@uab.es

Abstract. PCs and Workstations clusters are becoming more popular everyday. In many cases these systems are considered as part of a wider system for meta or GRID computing purposes. The management, administration and use of this systems present several difficulties due to the need to access different systems located at remote sites. A new architecture for managing a MultiUser MultiCluster environment has been developed. A cluster management environment prototype called ION Web (Interoperable Object Network Web) has been developed. This prototype is based on well-known standards (Web, CORBA, Java and XML). It works with PCs or Workstation Clusters but the architecture is not final environment dependent, and can be migrated or adapted to any cluster, like for example, PVM or MPI Clusters, or Grid systems using their native services.

1 Introduction

The use of parallel/distributed systems must be considered from different points of view depending on the kind of user involved. System administrators must take care of the system configuration, access control, application installation and so on. Application developers carry out the task of designing and building parallel/distributed applications, using a reduced set of practical APIs, like PVM[1], MPI[2], BSP, etcetera. Finally, end user runs applications, providing the input data and getting the output results. In the classical approach each one of these kinds of users have a complete different view of the system with a different user interface. Moreover, there are no uniform interfaces and each application must redefine its own interface. This fact causes the rebuilding of application runtime environments for data management, execution control and results publishing, for each system involved.

* This work has been supported by the MCyT (Spain) under contract TIC2001-2592 and partially supported by Generalitat de Catalunya – G. de Recerca consolidat 2001SGR-00218.

Our proposal is to define an architectural framework, n-tier based, that provides support to the transparent management/use of remote heterogeneous distributed (and parallel) systems. This proposal is based on providing an API that offers a set of basic remote services for the cluster system that can be used from any type of device (PCs, cellular phones, PDAs) or software clients.

The proposed architecture needs to provide different points of view of the Cluster/Parallel system, concerning the offered model to the architecture clients:

- End user needs a functional model of the system utilisation, which includes an abstract view of the services provided by the target system to the user.
- Administrator requires services for system configuration from a structural and functional point of view, and services to control the interactions of the users with the system.

In the literature several approaches have been described to design computing portals (WebFlow [3], Cactus[4], M3C[5]). However, these systems mainly care about the user interface, but not about offering some generic services since they take some existing platform to offer its services (for example, Grid systems [6]).

Our approach uses a n-tier model that separates the system modules in a reduced set of layers. A prototype called IONweb (Interoperable Object Network Web) based on well known object oriented technologies uses the layered architecture.

Section 2 describes the requirements of the system and the multilayer architecture proposed. Section 3 analyses the three main layers. Section 4 presents the implementation of a prototype called IONweb. Section 5 shows a case use. Finally, section 6 presents some conclusions and several possibilities of future development.

2 Web Remote Services Oriented Architecture

As it has been mentioned above, when considering a cluster system, there are three different kind of user with different degree of expertise, different purposes and, therefore, different requirements. These three types of “user” are system administrators, application developers and end users. So the first step to define the system architecture is to analyse the requirements of each one of these “users”.

From an end user perspective the system must provide the following facilities:

- Transparent access to the system. The user does not need to know the system features, their heterogeneities or problematic (if his application does not need to know them).
- Remote management of computing distributed/parallel systems (like clusters) for launching applications.
- End user can ask for: 1) Run installed applications; 2) Run applications with a personal configuration of the user (or predefined configuration); 3) Upload the application to be run (and the set of data and parameters needed); ACLs (Access Control Lists), The correct management of the computing system requires the creation of different user profiles with different capabilities. Therefore, it is necessary to establish Access Control Lists (ACL).

- End users can decide to run their applications in different modes: 1) Interactive, the user can be connected for launching and waiting for the task termination (in short tasks), or it may disconnect and after reconnect for consulting their tasks status. 2) Batch, launching in disconnected mode (in time or priority queue), one task or a pool of tasks. 3) Parameterised tasks, multiple runs of a task with variation of parameters (or data used).

Considering developers of applications the system should also provide the following facilities:

- Developers or applications testers need compiling native tasks.
- Remote terminal access for specialized tasks (like environment reconfiguration, sources or data edition).

From a system administration perspective the facilities should also include:

- Users Profiles: The system needs to manage different categories of user, for a restricted system view: information access permissions, resource allocation, services call restrictions...
- Virtual systems management: Creation of virtual environment of execution, partitioning and mapping of pseudo virtual machines available to different profiles of users.
- Management of Tasks Pools.

There is an additional type of user involved in the system. It can be called a system integrator (usually can be the system administrator). System integrators are interested in including new nodes or clusters in the distributed system.

These requirements involve different level of abstractions, some refer to final user interface, others to internal management, and others deal with services provided by the distributed computing systems.

The proposed architecture involves multiple layers:

- User Layer (UL): functional interface with users.
- Remote Service Interface Layer (RSIL): Interface interconnecting users to cluster computing services.
- System Controller Layer (SCL): Interface controlling communication with cluster, defining final services provided by the cluster system.
- System Layer (SL): The cluster systems.

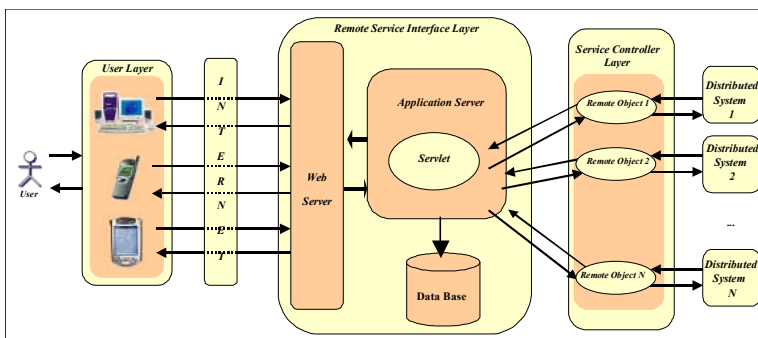


Fig. 1. General view of the multilayer architecture

Next section describes the three main layers of the proposed architecture: User Layer (UL), Remote Service Interface Layer (RSIL) and Service Controller Layer (SCL).

3 Architectural Layers

3.1 User Layer: Multiple Device Interface

Internet is providing multiple interconnection forms to the information systems. In this evolving interconnected world, users adapt their work to the available devices at every moment. Therefore, flexibility becomes an important feature of current systems.

Most applications running in a cluster system require some common management tasks such as configuring input data, configuring the execution parameters, launching the application, checking the execution status, obtaining data of application execution (results), etc.

These management tasks do not require great computing power, but only a specification how to carry out them and the data themselves. These tasks can be done by the actual power of more simplified devices, like for example cellular phones with enabled Internet capabilities (WAP phones), PDAs, or any other device of mobile computing. So, these devices can be used for these purposes and the user can be liberated from working on the target system itself and just accessing it through internet.

The User Layer (UL) provides the required independence and offers the interface facilities for treating all these devices. Most of these mobile devices have some enabled web browser, and the information pages (html pages) needed for management the system can be adapted and translated to display in these devices. For example, WML pages (for mobile phones) can be adapted from their corresponding HTML pages to manage the same operation. Web and XML [7] technologies provide data independent representation, and data independent visualization on multiple devices.

3.2 Remote System Interface Layer

The Remote System Interface Layer (RSIL) acts as intermediate layer between final user interface (UL) and the gateway to cluster systems (SCL). It provides the management logic to the operations available to the users (figure 2), mainly by means of remote cluster services.

Depending on the complexity of each operation, the translation can be direct (one operation to a service call), chained direct (a set of basic services provides one operation), or can require internal data management, for example when the cluster system does not provide such facilities (for example, user profile management, job/tasks/users status historic management).

This internal data management requires a storage data system, to integrate data information flows in the architecture. This data storage system must store the user initiated operations or in course, partial status information of operations, monitoring information, or system statistics.

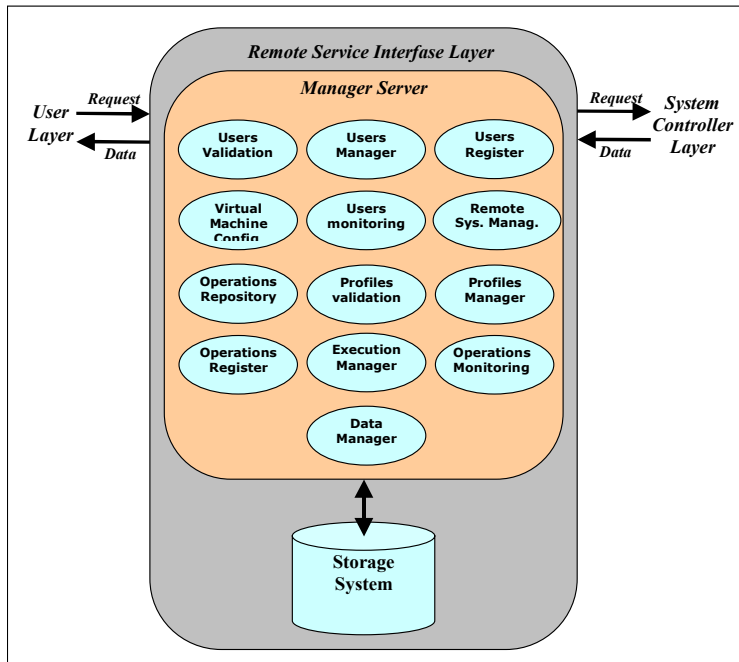


Fig. 2. RSIL management processes

3.3 Service Controller Layer: Cluster Remote Services Access

The Service controller layer (SCL) incorporates a gateway to access the services provided by the Cluster system from external device/system/Web Portal clients, isolating the cluster system from the external management system. This architecture provides a clear separation between the layers of user presentation, system logic and the final system. The cluster system and the provided services are independent of the hardware/software client platform.

The gateway is formulated as an API specification of a set of services offered to the clients. The gateway clients need a set of operations in the RSIL. The services are the basic indivisible operations, that can be done in the cluster. These services are used for operations such as Job configuration, Job Monitoring, File Upload/download, Virtual system configuration, System Status, User tasks status, System status, etc.

These services are provided in a standard form including I/O data and their formats. However, the services are not only static, but they can be specified dynamically. An static services API offers a reduced set of services with very generic interface that can be easily adapted to any target system. But the final system can provide particular added services, that must be offered to the client. This fact forces to open the possibility of specifying new services. In our cluster management system, this new added services specification can be done without recompiling or regenerating the gateway using XML service description.

4 Implementation of the IONweb Prototype

In our prototype IONweb, we have used well known standard technologies for developing the architecture layers. Figure 3 shows the technical solutions involved and their interconnections.

In the final interface to the user (UL), we were looking for independence from visualization system (PC, cellular phone, PDA). The use of XML technologies provides independence of data representation, and data visualization, by readapting data to view formats. The combination of XML data representation with XSL [8] transformations generates final HTML pages to different destinations. In our prototype, we have tested the generation of interface to Web browsers in standard form by HTTP protocol and HTML pages, and to WAP interface for mobile phones with WML. Other specialized devices (like PDAs), could be included by applying the necessary XSL styles to adapt de data outputs. Using our interface, the user can connect to the system using multiple devices to run, test or monitoring his/her tasks & applications.

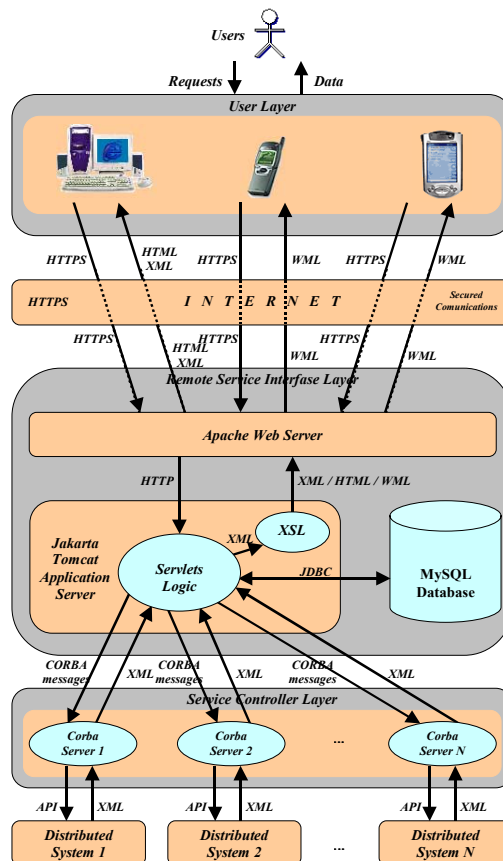


Fig. 3. IONweb prototype architecture

In the interface RSIL, we provide a web server based solution, with open standards. It includes a secured Apache Web server, with HTTPS protocol for users access to clusters. The layer logic is implemented by a combination of Java servlets and database access. The servlets encapsulate functionality of accessing to remote cluster services, and maintaining state information in a database: state of the cluster sessions, cluster tasks, users state, and control users rights (ACLs, access control list) in their profile for access the final services. The servlet runtime engine used is Jakarta Tomcat, and Java JDBC drivers for accessing the database system (MySQL).

In the SCL we deal with the final cluster system providing an external standard accessible services. In our prototype we implement this using CORBA [9] middleware technology. In CORBA, the server objects can be remotely activated and response to service requests. The services implemented are language, platform, and network independent, and can be accessed by means of a standard runtime called ORB (Object Request Broker), that manages server access and client requests. The used ORB implementation has been Iona ORBACUS.

In our prototype, we are working with PVM Clusters, and the SCL is implemented as a CORBA server dealing with communication to PVM daemons. Our Server can be initiated on a cluster node called representative node (or entry point). This CORBA server represents the cluster, and have a persistent internal configuration (written in XML), with information of cluster nodes and their capabilities.

5 Use of IONweb

In this section we will show a case use of an end user that wants to run a PVM application on a particular cluster. The end user connects to a web site and logs in using the interface offered by the UL. The RSIL validates the user, gets his/her user profile from the database and provides information to UL so that the possibilities available to the user are shown in a new page. The user selects, the cluster, the application, the data files and submits the application to the system. The UL transfer the request to the RSIL and this layer connects to the services interface in SCL to start up and launch the application on the cluster system. Now the application is running and the user can decide to logout and reconnect afterwards to check the status of his/her job, or to get the results.

6 Conclusions and Future Work

The system described provides an easy way to use and manage cluster systems from remote clients. The architecture provides different views of the target system to users, adapting services of the clusters to application users, application developers, cluster administrators or cluster systems integrators. The architecture is organised in three main layers, concerning user interface, management logic and cluster services.

The use of independent layers simplifies the adaptation to the evolving user interfaces provided by different client devices (PCs, mobile phones, or PDAs).

They are many ways to improve actual architecture, concerning new services in the system layer:

- Remote console service for clusters. This provides a web remote interface for accessing the cluster, improving typical user operations related to the processes of edition, compilation and customisation of an application.
- Remote visualization service. Data visualization is a very useful and essential for the application users. This can be integrated in the system using standard file downloading services, and plug-in browser technology for adapting them to known formats or creating a new viewer for the application.
- Service access protocol. Some new technologies, like SOAP protocol [10], could provide better performance results in the interlayer communications for accessing the cluster services. A more standard access API offering a new level of abstraction can be obtained using Web Services [11] paradigm. WSDL (Web Services Description Language) and UDDI (Universal Description Discovery and Integration) can be used for offering the architecture services in standard form, and facilitating the description and discovery of new cluster services.

References

- [1] Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, W., Mancheck, R. and Sunderman, V., "PVM : Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing", The MIT Press, Cambridge, (1994).
- [2] Gropp, W., Lusk, E., Doss, N., Skjellum, A. "A high performance, portable implementation of the MPI standard". *Parallel Computing*, 22 (6): 789-828, 1996.
- [3] Akarsu, E., Fox, G.C., Furmanski, W., Haupt, T. "WebFlow – High level programming environment and visual authoring toolkit for high performance distributed computing". In proceeding of Supercomputing 98, (1998).
- [4] Allen G., Benger W., Goodale, T., Hege, H., Lanfermann, G., Merzky, A., Radke, T., Seidel, E., Shalf, J. "Cactus Tools for Grid Applications". *Cluster Computing* 4, 179-188, (2001).
- [5] Geist, A., Schwidder, J. "M3C - An Architecture for Monitoring and Managing Multiple Clusters", 12th IASTED International Conference on Parallel and Distributed Computing and Systems, Las Vegas, Nevada (2000).
- [6] Foster and C. Kesselman (eds). *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufman Publishers, (1999).
- [7] Extensible Markup Language (XML), available at <http://www.w3.org/XML/>
- [8] Extensible Stylesheet Language (XSL), available at <http://www.w3.org/Style/XSL/>
- [9] The Common Object Request Broker: Architecture and Specification, available at <http://www.omg.org>
- [10] Simple Object Access Protocol (SOAP), available at <http://www.w3.org/TR/soap12-part0/>
- [11] Web Services Description Language (WSDL), at <http://www.w3.org/TR/wsdl>

Improving Flexibility and Performance of PVM Applications by Distributed Partial Evaluation^{*}

Bartosz Krysztop and Henryk Krawczyk

Gdańsk University of Technology
Faculty of Electronics, Telecommunications and Informatics
ul. Narutowicza 11/12, 80-952 Gdańsk, Poland
krysztop@pg.gda.pl
hkrawk@pg.gda.pl

Abstract. A new framework for developing both flexible and efficient PVM applications is described. We propose Architecture Templates Interface (ATI) that allows to control application granularity and parallelism. To ensure high application efficiency we extend partial evaluation strategy into domain of distributed applications obtaining Distributed Partial Evaluation (DPE). Both ATI and DPE were implemented using a new distributed programming language TL (Transformation Language) which allows to express as well an application code as its transformation (optimisation) procedures. Experimental results confirm high usability of the proposed methodology.

1 Introduction

The impressive growth of computer power does not eliminate performance problems of user applications. Moreover, a new extra requirement is taken into consideration — high application productivity. In other words designed applications should be flexible and efficient i.e. their execution time should be low for different parallel environments.

PVM leads to three layers architecture (denoted by P_1) that offers basic message-passing mechanism and process management routines (see Fig. 1). A sequential code of processes is mixed with data exchange and process management code. This makes program code not flexible, its development and maintenance for many applications is more difficult and error prone.

To facilitate software development graphical support is used (GRADE [1], EDPEPPS [2]). An architecture of an application is defined as combination of basic blocks (sequential processes) and mechanisms describing types of interactions between them. The main advantage of this approach is that the sequential code of the application and its distributed architecture can evolve in a semi-independent manner, what supports fast prototyping and simplifies performance tuning. We can easily control granularity of the application by changing number

^{*} The paper was partially sponsored by State Committee for Scientific Research (KBN) under grant 8 T11C 001 17.

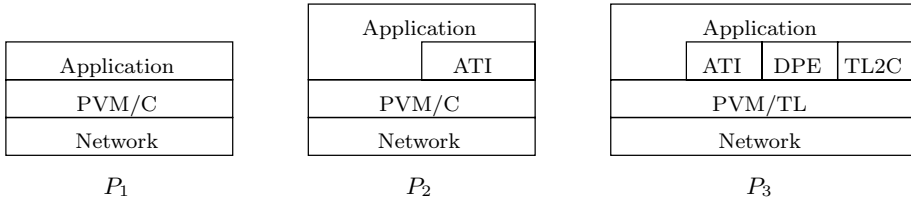


Fig. 1. Three considered programming platforms: P_1 traditional PVM platform, P_2 PVM extended with ATI, P_3 PVM extended with ATI, DPE and TL2C tools

of processes (for instance in master-slave model), however, *any radical change in application architecture requires rewriting sequential code to match a new decomposition*. For this reason management of parallelism at an arbitrary abstraction level is difficult.

We believe that controlling data decomposition and program granularity we can increase flexibility of application architecture. Therefore we introduce a new abstraction level called *Architecture Templates Interface* (ATI). ATI is responsible for implementing communication scheme between processes according to the selected processing model (see Fig. 1, P_2). However, high flexibility of applications leads to low performance, therefore we extend the well known optimisation technique called the partial evaluation (program specialisation) into distributed (parallel) one. The *Distributed Partial Evaluation* (DPE) tries to optimise the whole PVM+ATI application from performance point of view. Both ATI and DPE are implemented in TL (Transformation Language) environment [3, 4]. TL interpreter uses PVM for communication and process management (see Fig. 1, P_3). TL2C is the translator of a TL interpreted code into C+PVM one. It is used as the last stage of distributed application optimisation process.

Our goal was to check correlation between flexibility and performance in distributed PVM environment when developing software according to ATI paradigm with support of DPE optimisation process. In other words, we should state that applications implemented using P_1 platform (see Fig. 1) are efficient, but platform P_2 gives more flexible applications. Moreover, platform P_3 supports development of both efficient and flexible PVM applications. To measure quality of the proposed approach two main metrics are analysed: *relative execution cost* (*REC*) — describing efficiency and *relative cost of reused code* (*RCRC*) — describing flexibility.

In Sect. 2 we introduce ATI, the next section describes DPE procedure. Section 4 explains used quality metrics, presents experimental results and evaluates usability of the proposed technique for parallel software engineering. In the last section we refer our approach to other techniques described in the literature.

2 Architecture Templates Interface (ATI)

ATI hides from the user all mechanisms connected with processes management and data distribution implemented in the parallel applications. It allows to specify how data should be processed by specifying pieces of code that process data and interactions (dependencies) between them. By providing single boolean flag (“working mode”) we can easily control if given codes are executed in the same PVM process (sequentially) or in parallel. ATI takes care of dynamic process creation and manages proper communication scheme according to user specifications. Data decomposition and functional decomposition impacts on performance and scalability of distributed algorithms. The master-slave architecture with data decomposition and pipeline with functional decomposition are provided by ATI.

TL interpreter [4] is build on the top of PVM system and allows to execute distributed applications. Details of its syntax and semantic can be found in [3]. The language allows to perform runtime analysis and synthesis of executable code what is useful for implementing advanced optimisation and transformation procedures (such as DPE). TL increases flexibility of an application code by utilisation of code reuse mechanisms and higher order abstractions (templates).

3 Distributed Partial Evaluation (DPE)

A partial evaluation (PE) is the optimisation procedure that specialises given sequential code according to constant (known at optimisation time) parameters. For example C code: `b=3;c=x*(b+5);` can be specialised (reduced) to: `c=x*8;`. PE approach is a well known technique that has been investigated for decades [5]. However, the majority of described in literature solutions are limited to the domain of functional languages where partial evaluation is straightforward. Relatively few papers describe application of partial evaluation for imperative languages (such as C or Fortran). A survey of them can be found in [6]. Partial evaluation was also analysed with respect to concurrent languages involving asynchronous data exchange between processes [7, 8]. However, proposed solutions are usually limited to CSP-like languages with very simple constructs rather not suitable for real life implementations. We investigated usability of PE methods for TL with message passing and dynamic process creation in a distributed environment. In consequence we offer Distributed Partial Evaluation (DPE) approach.

To handle partial evaluation of TL code we described it by *expressions* (basic and complex ones that consist of sub-expressions) similar to the ones in functional languages. We distinguish *three* types of expressions: 1. *r-expression* stands on the right-hand side of assignment operator (its r-value is significant), 2. *l-expression* stands on the left-hand side of assignment operator (its l-value is significant), 3. *n-expression* does not evaluate to meaningful value (only side effects are observed). In TL the PE algorithm uses the divide and conquer approach. For each complex expression, all sub-expressions are optimised (in order

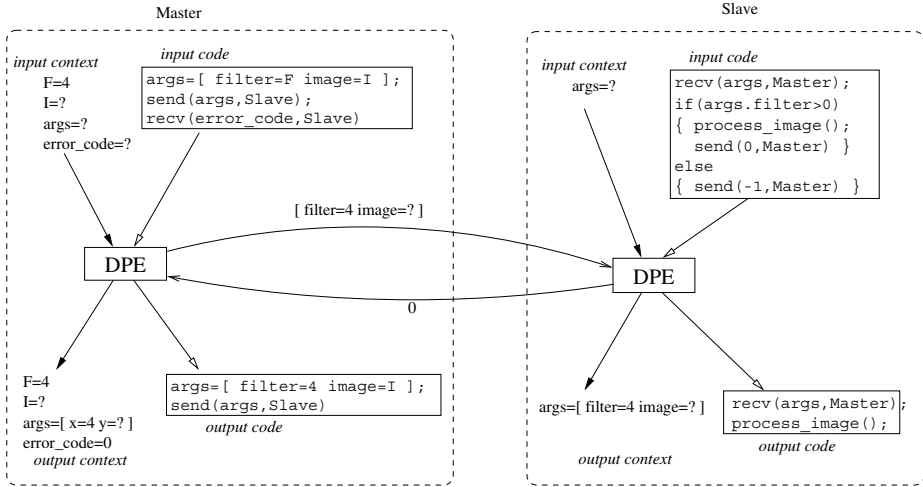


Fig. 2. Distributed Partial Evaluation (DPE) with message passing

of original evaluation) and results of the optimisation are used to build optimised code for the complex expression. In this way the optimisation algorithm “goes through the code” in the same order as TL interpreter what guarantees proper order of assignments and the same side effects. Since, in general it is not possible to predict, in what order the optimised sub-expressions will be used for construction of the result, three types of optimised expression described above are taken into consideration. Computational complexity of PE is $O(n)$ where n is number of steps performed during execution of original code [3].

To handle distributed applications we had to define behaviour of DPE procedure with respect to dynamic process creation, management and communication expressions. To do this we made the following assumptions:

- dynamic process creation does not depend on input data (number of processes to create and their codes),
- each concrete communication scenario described by number of messages, their sources and destinations does not depend on input data — this is required to control data flow between processes.

Then, the DPE procedure [4] is stated as follows:

1. the sequential partial evaluations (PEs) for each process of the PVM application are performed in parallel,
2. communication between optimised processes is the same as in the original application, however, messages may contain incomplete data (only static values, known at the time of optimisation),
3. all messages that contain complete data are removed from the optimised application during partial evaluation, the others are left, like in the original application.

Illustration of DPE execution for an application with two processes is given in Fig. 2. More detailed description of distributed partial evaluation can be found in [3].

To obtain highly efficient code a TL application should be translated into C. This has been realized by a TL2C translator (implemented in TL, see Fig. 1, P_3). By using DPE+TL2C tools, an original (reusable) TL code consisting of many components (user code, ATI library etc.) is transformed into the monolithic code with performance comparable to manually written PVM code of the same application.

4 Experiments

To evaluate the proposed approach a number of experiments took place. Below we present some representative results obtained for the image processing application. The application processed a set of images, each image was transformed in two steps: image transformation (convolution based edge detection) and calculation of histogram of the transformed images.

In such experiments we concentrated on three programming platforms and utilised five processing models. According to Fig. 1 the following platforms were analysed:

1. efficiency oriented PVM/C (P_1), all manual implementations concentrated on high execution efficiency of the applications,
2. flexibility oriented ATI/PVM/C (P_2), all implementations concentrated on minimal modification costs of the applications with an acceptable level of its efficiency,
3. combined approach (P_3), where applications have been implemented in TL using ATI library, optimised by DPE procedure and translated into C+PVM code by TL2C.

The following processing models determined by application architectures were considered:

- sequential (M_{seq}), consisted of a single process,
- master-slave, where each slave sequentially transforms image and calculates histogram, this architecture was implemented in two variants: with input images placed at either at root node ($M_{m-s(1)}$) or at slave nodes ($M_{m-s(2)}$),
- pipeline consisted of two stages, where the first stage uses master-slave model for edge detection, and the second stage (histogram calculation) is executed as a single process. The architecture was implemented also in two variants: with input images placed either at root node ($M_{pipe(1)}$) or at slave nodes belonging to the first stage ($M_{pipe(2)}$).

We also considered two versions of image processing: grayscale or colour images. In consequence we obtained 30 different implementations of the application where each implementation has the same functionality but different code. We

Table 1. *REC* and *RCRC* as function of programming platforms and application architectures

| | <i>REC</i> [%] | | | | <i>RCRC</i> [%] | | |
|---------------|----------------|----------|--------|---------------|-----------------|-------|-------|
| | P_1 | P_2 | P_3 | | P_1 | P_2 | P_3 |
| M_{seq} | 100 | 2450±632 | 143±1 | M_{seq} | 84.90 | 69.61 | 49.60 |
| $M_{m-s(1)}$ | 100 | 292±182 | 105±4 | $M_{m-s(1)}$ | 70.11 | 45.99 | 46.71 |
| $M_{m-s(2)}$ | 100 | 1212±399 | 125±17 | $M_{m-s(2)}$ | 68.80 | 45.52 | 52.81 |
| $M_{pipe(1)}$ | 100 | 144±42 | 102±7 | $M_{pipe(1)}$ | 65.68 | 40.98 | 44.97 |
| $M_{pipe(2)}$ | 100 | 143±31 | 99±4 | $M_{pipe(2)}$ | 64.73 | 40.86 | 50.03 |

used PVM environment consisting of 6 Sun workstations. During single execution each program processed 60 images. Experiments were performed for various image sizes and different number of slave processes changing from 2 to 6.

Experiments have shown that P_1 implementations are characterised by the shortest execution time for almost all cases. We used them as base reference for efficiency evaluation of other platforms via *relative execution cost* metric (*REC*):

$$REC = \frac{t_x}{t_{P_1}} \times 100\% \quad (1)$$

where:

- t_{P_1} — execution time of P_1 implementations,
 t_x — execution time of other implementations (P_2 or P_3).

Both execution times referred in (1) are measured under the same conditions (architecture, input data size, number of slaves etc.). Table 1 summarises the mean *REC* values for different approaches with standard deviations given after \pm sign.

To evaluate code flexibility we used *relative cost of reused code* metric (*RCRC*) [9], which can be expressed via number of source lines:

$$RCRC = \frac{LOC(X \rightarrow Y)}{LOC(Y)} \times 100\% \quad (2)$$

Table 2. *QF* for different programming platforms and application architectures

| | P_1 | P_2 | P_3 |
|---------------|-------|-------|-------|
| M_{seq} | 1.17 | 0.05 | 1.40 |
| $M_{m-s(1)}$ | 1.42 | 0.74 | 2.03 |
| $M_{pipe(1)}$ | 1.45 | 0.18 | 1.51 |
| $M_{m-s(2)}$ | 1.52 | 1.69 | 2.18 |
| $M_{pipe(2)}$ | 1.54 | 1.71 | 2.01 |

where:
 $LOC(X \rightarrow Y)$ — number of lines added/changed when converting code X
 into code Y of the same application,
 $LOC(Y)$ — number of lines in code Y .

Table 1 contains also mean values of *RCRC* for all considered program modifications. According to obtained results P_3 implementation appears to be the most flexible. This becomes obvious if we realize that parallelism can be controlled via single “working mode” parameter provided to ATI (see Sect. 2). One of the surprising results is that ATI implemented in C does not increase flexibility (platform P_2), this can be explained by nature of C language (efficiency oriented without support for higher order abstractions, strong code reuse etc.).

In order to summarise obtained results modification and execution costs can be quantified giving common *quality factor* (QF) which we define as follows:

$$QF = \frac{1}{RCRC \times REC} \tag{3}$$

Table 2 shows that the highest QF values are observed for P_3 approach what confirms its ability to combine flexibility and high efficiency in PVM applications. Correlation between performance and flexibility of all experimental applications is illustrated in Fig. 3. A single point stands for one application execution. Its location is determined by application relative execution time (*REC* metric) and average modification cost (*RCRC* metric). Points placed closer to the beginning of coordination system have larger QF value.

5 Conclusions

We described approach that can reduce time and cost required to develop highly flexible and efficient PVM applications. It assumes that parallelism is controlled via generic ATI. We extended traditional notion of partial evaluation to handle distributed applications obtaining inherently distributed optimisation procedure DPE. The procedure eliminates negative impact of extra ATI layer onto application execution efficiency. Experimental results show that the proposed approach does not reduce performance significantly in comparison to manually written PVM applications, however, it offers more flexible architecture. Table 3 summarises selected aspects of existing approaches [1, 2] compared with our

Table 3. Comparison of modified PVM programming platforms

| platforms | graphical tools | C/PVM (P_1) | ATI/C/PVM (P_2) | TL/ATI/DPE/PVM (P_3) |
|----------------------|--------------------|--------------------|------------------------|-----------------------------|
| control granularity | + | - | + | + |
| control architecture | - | - | + | + |
| high efficiency | + | + | - | + |

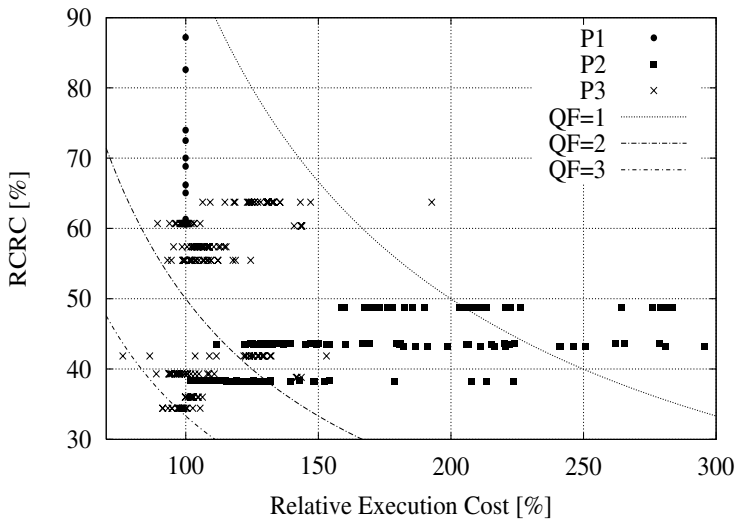


Fig. 3. Relative execution time (*REC*) and modification cost (*RCRC*) for different implementations of the same application.

approach. The platform P_3 appears to be promising approach, but its real evaluation requires more investigation with other classes of PVM applications.

References

- [1] Cunha, J., Kacsuk, P., Winter, S.: Parallel Program Development For Cluster Computing. Nova Science Publishers, Inc. (2001) 376, 382
- [2] Delaitre, T., Justo, G., Spies, F.: Winter: A graphical toolset for simulation modelling of parallel systems (1997) 376, 382
- [3] Krysztop, B.: Software integration by code transformations ensuring high modifiability and efficiency of user applications. PhD thesis, Gdańsk University of Technology (2002) 377, 378, 379, 380
- [4] Krawczyk, H., Krysztop, B.: A new approach for increasing efficiency and flexibility of distributed applications. In: Proceedings of the 27th EUROMICRO Conference (EUROMICRO 2001). (2001) 192–199 377, 378, 379
- [5] Futamura, Y.: Partial evaluation of computation process – an approach to a compiler-compiler. Systems, Computers, Controls 2 (1971) 45–50 378
- [6] Beckmann, O.: Partial evaluation, imperative languages and c (1996) 378
- [7] Marinescu, M., Goldberg, B.: Partial-evaluation techniques for concurrent programs. In: Partial Evaluation and Semantic-Based Program Manipulation. (1997) 47–62 378

- [8] Hosoya, H., Kobayashi, N., Yonezawa, A.: Partial evaluation scheme for concurrent languages and its correctness. In Bougé, L., et al., eds.: Euro-Par'96 – Parallel Processing, Lyon, France. (Lecture Notes in Computer Science, vol. 1123), Berlin: Springer-Verlag (1996) 625–632 [378](#)
- [9] Krawczyk, H., Wiszniewski, B.: Analysis and Testing of Distributed Software Applications. Research Studies Press Ltd. Baldock, Hertfordshire, England (1998) [381](#)

Ready-Mode Receive: An Optimized Receive Function for MPI

Ron Brightwell

Scalable Computing Systems, Sandia National Laboratories
Albuquerque, NM 87185-1110, USA
`bright@cs.sandia.gov`

Abstract. This paper describes an extension to the MPI Standard that offers the potential to increase performance for receiving messages. This extension, the *ready-mode receive*, is the receive-side equivalent of the ready-mode send. This paper describes the semantics of this new receive function and discusses the potential for performance improvement that it offers. In particular, we discuss how the current trend toward using intelligent network interfaces may increase the potential for significant performance improvement.

1 Introduction

In February of 1996, the author made an informal proposal to other members of the MPI Forum for a new receive function that offered the opportunity for increased performance. At the time, the MPI Forum was working on additions and changes to version 1.0 of the MPI Standard [1], which became MPI version 2.0 [2]. The initial reaction to this proposal was largely negative, so a more formal proposal with supporting evidence was never pursued. However, we believe that the proposed receive function warrants further investigation for several reasons. In this paper, we describe the proposed function and discuss the possible benefits it offers. We also discuss some of the arguments that were presented against including the function in the Standard.

The rest of this paper is organized as follows. The following section describes the new receive function in detail and discusses the possible performance implications surrounding it. Included in this section is a summary of the initial arguments against this function from other members of the Forum. Section 3 continues with a rebuttal of these arguments and presents some further motivations for investigating the possible benefits of the new receive operation. We conclude by summarizing the key points of this paper in Section 5 and provide an outline of future activities in Section 4.

2 The Ready-Mode Receive

The MPI version 1.2 Standard defines four different modes for sending data. Each of these modes has different semantics and associated with moving data that affect how the underlying MPI implementation handles buffering and completion.

The *standard*-mode send operation has no semantic guarantees to buffering or completion. The MPI implementation is free to use any buffering strategy (e.g. send-side or receive-side) and completion may or may not be tied to the existence of a matching posted receive at the destination process. A *buffered*-mode send is also free to complete independent of activity at the receiver, but in this case, the sending process has given the MPI implementation sufficient space to buffer the outgoing message should it need to. The *synchronous*-mode send does not complete until a matching receive operation has been initiated at the destination. In addition to data movement, this send mode provides an explicit synchronization point between the processes. Finally, the *ready*-mode send may not be started until a matching receive has already been posted at the destination. This mode offers the opportunity for increased performance. Since the receive is guaranteed to be posted, the sender can make optimizations with respect to avoid buffering and avoiding handshake protocols with the receiver.

The ready-mode send is the only mode whose semantics are driven by performance. It is an indication from the application programmer of an opportunity to avoid protocol and buffering overhead in the interest of increased performance. The possible benefits of ready-mode is fairly straightforward. Eliminating intermediate buffering, especially for large messages, saves resources and avoids the performance degradation of memory-to-memory copies. Avoiding handshaking protocols eliminates the extra overhead of communicating with the destination process before moving the data. To some extent, these costs are relatively easy to measure.

In contrast to modes for sending, there are no equivalent receive modes defined by MPI for the semantics associated with message reception. The semantics of a receive operation are determined by the matching send operation. For example, a receive operation that matches with a synchronous mode send needs to generate an acknowledgment to the sender in order for the send to complete. A receive operation that matches with a standard mode send may not need to do anything to complete the send. Since the semantics of standard-mode, buffered-mode, and synchronous-mode are all defined by message completion, it is appropriate for the receive operation to determine what if anything needs to be done after a matching message has been found.

However, this is not the case for the receive operation that matches a ready-mode send. For this operation, the semantic is defined by message initiation. That is, the ready-send cannot start until a matching receive has been posted. Similar to the optimization opportunity of a ready-mode send, the corresponding receive operation that matches a ready-mode send has a possible opportunity for optimization. Since the semantics of ready-mode send guarantee that a matching receive has been posted at the destination, the matching receive operation is guaranteed that no matching message has yet arrived. For a *ready-mode receive*, there is no need to search a queue of unexpected messages for a possible match.

2.1 Potential Benefits

The obvious potential benefit of the ready-mode receive is avoiding the time needed to search an unexpected message queue before posting the receive. This time can be dependent on several factors, but primarily depends on the average length of the unexpected queue and the cost associated with traversing it. This information is highly dependent on the MPI implementation and the message passing structure of the application. The performance increase may be significant for MPI implementations where the cost of searching an queue is relatively high or for applications whose unexpected queue can grow relatively long. Unfortunately, no real data for these two measurements is readily available or published.

For some implementations, searching an unexpected queue may have side effects beyond the time wasted to perform the operation. For example, some MPI implementations (e.g. [3, 4]) maintain two structures for maintaining an unexpected message queue. Part of the queue is maintained by the network interface and part is maintained inside the MPI implementation. For our MPI implementation for Portals 3.0 [4], events related to unexpected messages must be maintained in two separate queues. Message receive events are placed in a queue by the network interface and consumed by the MPI implementation. If MPI consumes an event from this queue that does not match the current receive operation, it must hold on to this event in a separate queue. This could potentially result in several unneeded memory-to-memory copies of message header information and possibly data.

Since searching an unexpected queue and posting a receive must be an atomic operation, the MPI library must be extremely careful to retain atomicity. As such, the time needed to post a receive is dependent on the frequency of message arrival. If a message arrives after the unexpected queue is search, but before the receive is posted, this new message must be checked to insure that it does not match the receive. If messages are continually coming in while a standard receive is being posted, the post will be delayed until all of the receives have been made visible to MPI and the network has quiesced.

The performance of a ready-mode receive should be deterministic. Adding an entry to a posted receive queue should take a fixed amount of time. This is unlike the standard receive mode, which must traverse an arbitrarily long queue and possibly exchange protocol messages with the sender. This deterministic behavior may be beneficial to applications where the exchange of messages is more tightly synchronized, such as in some soft real-time applications. Deterministic performance of posting a receive may also enhance performance debugging by making detection of performance anomalies easier.

Since the ready-mode receive is a fundamental operation, implementations which do not have support for it can simply use the standard receive operation. The ready-mode receive offers an optimization opportunity, but does not sacrifice portability or correctness for those implementations that do not take advantage of the opportunity.

2.2 Drawbacks

A proposal for a ready-mode receive function was never formally brought to the MPI Forum. Initial feedback on the MPI mailing list was largely negative, with a few members adamantly opposed to such a function. Several reasons were cited.

First, the opportunity for performance is not readily evident or easily quantifiable. No data was available to support the claim that the ready-mode receive could provide a significant performance improvement. It was believed that high-performance applications, at least those that would be concerned about saving microseconds by not searching a queue, would have relatively short unexpected message queues. And, if the unexpected queue does grow to a point where search time would be significant, the MPI implementation should consider using a hashing function to avoid a linear search. This approach would increase the time to insert the unexpected into the queue, but would reduce the time needed to search the queue.

It was also pointed out that ready-mode sends are an optimization intended to address long messages, so saving a few microseconds for a communication that takes orders of magnitude longer is of little or no gain.

As for deterministic performance, it could be argued that for applications whose communication patterns are relatively consistent, the number of unexpected messages is likely to be low. An application that would benefit from the deterministic performance of a ready-mode receive is unlikely to have an unexpected queue long enough to cause a large variance in the time needed to search the queue.

Aside from the technical reasons related to performance, there were other reasons not to consider the ready-mode receive function. Those members of the MPI-2 Forum who participated in the MPI-1 Forum indicated that the ready-mode send mode was accepted into the Standard by a very close vote. Many on the Forum did not see any real performance benefit for ready-mode, and in an effort to try to keep the number of functions for data movement minimal, did not support it in the first place. The ready-mode receive function was viewed as a further optimization for a mode that was largely unsupported.

Lastly, it was argued that an additional receive function would be too confusing for application developers. In 1996, application developers had limited experience with MPI, and any semantic change to the basic data movement operations was viewed as an additional obstacle for application developers. The obvious drawback is that incorrect use of the ready-mode receive function would cause non-compliant programs and result in undefined behavior.

3 Current Motivations

At this point we have discussed the potential benefits and drawbacks of a ready-mode receive mode. We believe some of the original arguments against have weakened over time.

The important question of the performance benefits still remains open. The need for empirical evidence remains. We believe that the need for investigation of the ready-mode receive has grown due to the evolving networking technology to which the current generation of MPI implementations is targeted.

Commodity cluster computing has displaced proprietary parallel systems as the most popular platform for high-end scientific and engineering computing. Gigabit networking technology that utilizes intelligent or programmable network interface cards, including Myrinet [5], Quadrics [6], and VIA [7], have characteristics that may increase the cost of unnecessarily traversing the MPI unexpected queue. These network cards currently all use a PC's PCI bus, which is a significant bottleneck to achieving network performance. High-performance message passing layers typically avoid crossing the PCI bus whenever possible. Unnecessarily searching the MPI unexpected queue may involve a significant amount of traffic on the PCI bus. This may be especially true for implementations that offload MPI functionality onto the network interface card, a practice which is becoming more common as the computational power and memory capacity of network interfaces continues to increase.

We also believe that advanced MPI implementations will continue to move more data movement functionality from the user-space library down to intelligent or programmable networking hardware. As this happens, applications seeking the highest levels of performance will migrate toward those functions in MPI that offer greater opportunity for optimization, such as the ready-mode send and persistent communication mode.

In our experience developing and supporting MPI implementations for large-scale parallel machines, we have seen applications that require a significant number of unexpected messages. In fact, we have had to enhance an implementation solely on the basis of being more flexible in supporting larger numbers of unexpected messages, especially as applications are scaled up to thousands of processes. The number of unexpected messages increases with the number of processes in the job. We have seen applications that have exceeded 1024 unexpected messages on only a few hundred processors. Perhaps it can be argued that such applications are poorly designed or structured. It may be precisely this type of application that could benefit from restructuring using ready-mode send and receive functions.

In addition to direct usage by applications, the ready-mode receive function has many possible uses in supporting current MPI functionality or other possible extensions to the Standard.

One possible use in the MPI-2 one-sided operations. A possible portable implementation of the `MPI_Win_get()` function would be to post a receive and send a request message to the target. In this case, there would be no reason to search the unexpected queue to see if a matching message has already arrived. This optimization may decrease the latency of the get operation, especially in the case of a non-blocking get.

Some collective operations may be able to avail of the ready-mode receive as well. For example, `MPI_Allreduce` and `MPI_Reducescatter` functions may be

staged so such a receive needs to be posted before a request or contribution of data is given. Avoiding a search of the unexpected queue in this case may not provide any performance improvement since all collectives are currently blocking operations. Should future versions of MPI support non-blocking collectives, a ready-mode receive may be more effective. Nevertheless, since the ready-mode receive is a more fundamental operation, it may provide other benefits beyond raw performance to these collective operations.

4 Future Work

We believe that the ready-mode receive function may offer performance gains for applications using the ready-mode send. We also believe that unnecessarily searching the MPI unexpected queue may have resource management side effects that may ultimately affect performance and/or effective use of resources. We intend to explore both of these issues in more depth and hope to provide experimental results that can be used to determine the effectiveness of the proposed receive operation.

Ideally, we would be able to find a real-world application that can be used to characterize the performance benefits of the ready-mode receive. However, we also intend to gather data related to the average length of the MPI unexpected message queue for different applications on various numbers of nodes.

A standard API or extension for gathering low-level performance data from within an MPI implementation is currently being developed as part of a research project sponsored by the United States Department of Energy's Accelerated Strategic Computing Initiative (ASCI). This work is being carried out in collaboration with MPI Software Technology, Inc., Pallas GmbH, and Intel KAI. This portable interface is intended to provide access to low-level performance data inside an MPI implementation, such as the length of the unexpected queue and the time a message has waited in the unexpected queue. We hope to leverage this interface to gather particular information that relates to the ready-mode receive.

5 Summary

In this paper, we have proposed an extension of the MPI Standard to support a new receive function, the ready-mode receive. This new function provides an opportunity for improving performance by avoiding the unnecessary traversal of an MPI unexpected message queue. We have discussed many advantages of this new function and also presented many arguments that do not support it. We believe that more in-depth analysis is needed to adequately evaluate the effectiveness of a ready-mode receive function. We intend to use a new interface that is being developed for gathering low-level MPI implementation data to more fully understand the implications and impact of this new receive function.

References

- [1] Message Passing Interface Forum: MPI: A Message-Passing Interface standard. The International Journal of Supercomputer Applications and High Performance Computing **8** (1994) 385
- [2] Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. (1997) <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html> 385
- [3] Dimitrov, R., Skjellum, A.: An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In: Proceedings of the Third MPI Developers' and Users' Conference. (1999) 15–24 387
- [4] Brightwell, R., Maccabe, A. B., Riesen, R.: Design and Implementation of MPI for Portals 3.0. (Submitted for Consideration to EuroPVM/MPI 2002) 387
- [5] Boden, N., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., Su, W.: Myrinet-a gigabit-per-second local-area network. IEEE Micro **15** (1995) 29–36 389
- [6] Petrini, F., chun Feng, W., Hoisie, A., Coll, S., Frachtenberg, E.: The Quadrics Network: High-Performance Clustering Technology. IEEE Micro **22** (2002) 46–57 389
- [7] Compaq, Microsoft, and Intel: Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel (1997) 389

Improved MPI All-to-all Communication on a Giganet SMP Cluster

Jesper Larsson Träff

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
`traff@ccrl-nece.de`

Abstract. We present the implementation of an improved, almost optimal algorithm for regular, personalized all-to-all communication for hierarchical multiprocessors, like clusters of SMP nodes. In MPI this communication primitive is realized in the `MPI_Alltoall` collective. The algorithm is a natural generalization of a well-known algorithm for non-hierarchical systems based on *factorization*. A specific contribution of the paper is a completely contention-free scheme not using token-passing for exchange of messages between SMP nodes.

We describe a dedicated implementation for a small Giganet SMP cluster with 6 SMP nodes of 4 processors each. We present simple experiments to validate the assumptions underlying the design of the algorithm. The results were used to guide the detailed implementation of a crucial part of the algorithm. Finally, we compare the improved `MPI_Alltoall` collective to a trivial (but widely used) implementation, and show improvements in average completion time of sometimes more than 10%. While this may not seem much, we have reasons to believe that the improvements will be more substantial for larger systems.

1 Introduction

The collective communication operations of MPI, the *Message Passing Interface* [12, 2], make it possible to express parallel computations on a more abstract level than by relying solely on point-to-point (or one-sided) communication, while still maintaining a high degree of portability. A high-quality MPI implementation will offer the best possible algorithms for each collective, carefully adapted to the underlying architecture. In this paper we describe and evaluate an algorithm, originally developed in [11] (which gives an abstract description and correctness proof), for the important `MPI_Alltoall` collective. The algorithm addresses specific features in current, hierarchical architectures like clusters of SMP nodes.

There has been a fair amount of work on algorithms for (the easy) collectives like barrier synchronization and broadcast for systems with a hierarchical communication system [7, 8, 9, 6], but little work has been reported on efficient all-to-all communication in this context; one exception is [5]. In this paper we describe an algorithm for `MPI_Alltoall`, which addresses the limited communication capabilities between SMP nodes. We validate the assumptions by

concrete measurements on a small SMP cluster with Giganet interconnect [1]. The resulting new implementation of the `MPI_Alltoall` collective for large messages sometimes performs more than 10% better than the trivial (but widely used) implementation. Although the improvements may look modest, we think they are worthwhile, and have reasons to believe that the algorithm will give more significant speed-ups on larger hierarchical systems.

2 The Hierarchical Factor-Algorithm

Given a set of processes, in MPI represented by a communicator `comm`. The `MPI_Alltoall` communication problem is the following: each process in `comm` has the same amount of data destined to every other process (including itself) in `comm`. Thus each process has to receive the same amount of data from every other process. In other words, every process pair has to exchange the same amount of data. This problem is sometimes referred to as *regular, personalized all-to-all communication*.

The problem can trivially be solved (and often is) by issuing the corresponding non-blocking send and receive operations, as in the following program fragment; `size` from now on denotes the number of processes in `comm`. Informally, we assume that data to be sent and received are located in the arrays `sendbuf` and `recvbuf`, respectively.

```
for (i=0; i<size; i++) {
    MPI_Isend(sendbuf[i], ..., i, comm, &request[2*i]);
    MPI_Irecv(recvbuf[i], ..., i, comm, &request[2*i+1]);
}
MPI_Waitall(2*size, request, status);
```

The problem with this algorithm is that the scheduling of the exchange is completely left to the runtime system. It can lead to network conflicts (contention) by many processes simultaneously trying to send to the same process. It can also lead to starvation, by processes in some time instant not finding a partner to/from which they have not yet sent/received data.

These drawbacks can be avoided by explicitly scheduling the communication by pairing processes, stepping through all possible pairings, and doing as many exchanges as possible concurrently. It is well-known that it is possible to do with `size` exchange rounds. In graph-theoretical terms, this is because the complete graph is *1-factorizable*. A particularly simple factorization is the following. Let the processes be numbered from 0 to $n - 1$, $n = \text{size}$. The partner of process u for round i is $(i - u) \bmod n$. In round $2u \bmod n$ process u is paired with itself. Another well-known factorization moves all self-loops to the last round for even n [4].

We call the algorithm based on factorization the (*flat*) *factor-algorithm*. It has been discovered many times, see for instance [3, 13, 10]. If we assume that function `factor` returns the partner of process $u = \text{rank}$ for round i in a complete graph of n nodes, we can write the algorithm as follows:

```

for (i=0; i<size; i++) {
    j = factor(rank,i,size); /* partner for round i */
    if (j==rank) copyself(&sendbuf[j],...,&recvbuf[j],...,comm);
    else          exchange(rank,j,sendbuf,recvbuf,comm);
}

```

The function `copyself` takes care of the self-loops, and copies data from send buffer to receive buffer. The `exchange` of data between process `rank` and its partner `j` can be implemented by a `MPI_Sendrecv` operation. This makes it possible to exploit bidirectional (full-duplex) communication, if the network allows this. The flat factor-algorithm is *optimal* for fully connected networks where all processors can communicate with each other at the same cost.

This assumption does not hold for SMP-clusters. Processes within a node can communicate via shared memory, typically faster, and concurrent operations are possible (to some extent, depending on the power of the memory system). Communication between nodes often has lower bandwidth, and not all processors can communicate concurrently. We make the assumption of *bidirectional, one-ported communication* between nodes, meaning that communication between two nodes at any one instant is limited to two pairs of processors communicating in opposite directions. This corresponds to the fact that SMP nodes typically have only one (or a small, fixed number of) network card(s). We assume that the inter-node network is homogeneous, such that communication between pairs of nodes is equally costly.

With these assumptions, we can now generalize the flat factor-algorithm to a *hierarchical factor-algorithm* [11]. We use factorization to pair nodes. When two nodes are paired, the processes exchange part of their data. Nodes may have different number of processes, and if paired nodes exchange all their data, “large” node pairs will dominate the time spent in a round, and we will not be able to obtain the result stated in Theorem 1 (see below). The hierarchical factor algorithm instead works in *phases*, in each of which a set of *active* nodes are paired over a number of rounds. We impose an ordering on the active nodes. Nodes $i \prec j$, if either the number of processes on node i is smaller than the number of processes on node j , or if i and j have the same number of processes and $i < j$ in some arbitrary but fixed ordering.

To sketch the algorithm we let i be the local index of each currently active node. For each node, `nodesize` is the number of processes on the node. For each phase `min` is the number of processes on the “smallest” active node. After the phase all nodes of size `min` will have completed their exchanges and become inactive.

```

active = nodes; A~ = [0,...,nodes-1]; /* active nodes */
min = 0;
while (active>0) {
    /* phase */
    done = min; /* nodesize done in previous phase */
    i = index_of_node(A); /* 0<=index of active node<active */
    min = minsize(A,active); /* size of smallest active node */
    for (round=0; round<active; round++) {
        j = factor(i,round,active);
        if (i==j) nodecopyself(done,min,i,sendbuf,recvbuf,comm);
        else      nodeexchange(done,min,i,j,sendbuf,recvbuf,comm);
    }
    if (nodesize==min) break; /* small node done */
    active = remove(A,...,min); /* remove small nodes */
}

```

When node i is paired with itself, the processes with local index between `done` and `min` exchange data with the processes with local index between `done` and `nodesize` on the node. This is done by the `nodecopyself` function. The exchange between nodes $i < j$ is done by the `nodeexchange` function: processes with local index between `done` and `min` on node i exchange their data with *all* processes on node j . Because of the (possibly) asymmetric `nodeexchange`, all exchanges within a round takes the same time (except for the `nodecopyself` exchanges, which are faster), and we obtain the following theorem (see [11] for proof).

Theorem 1. *The hierarchical factor algorithm correctly solves the regular, personalized all-to-all problem. All nodes with the maximum number of processes are busy throughout all phases. With a constant number of processors per node, the algorithm is asymptotically optimal as the total number of processors increase.*

3 Communication Capabilities of the SMP Cluster

We performed measurements on our Giganet cluster (for more details, see [1]) to validate the assumptions on intra- and inter-node bandwidth, and one-ported communication capabilities. The observations are used to guide the implementation of the `nodeexchange`. The most important communication patterns considered are shown in Figure 1. Each node has 4 processors (small circles); a line between two processors indicate exchange of data: each process sends and receives the same amount of data. Running times for the exchange of 512KBytes per process pair are given in Table 1.

We make the following observations. The difference between inter-node and intra-node bandwidth is marked, but not extreme (Pattern 1 versus Pattern 4). The bandwidth for concurrent intra-node communication drops by a factor of about 1.5 (Pattern 1 versus Patterns 2 and 3). The high bandwidth for inter-node

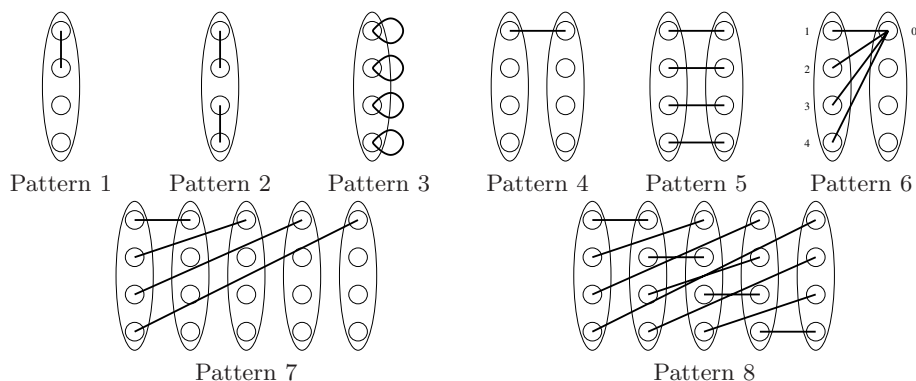


Fig. 1. Patterns used to investigate the communication capabilities of the Giganet cluster

communication can be sustained for one pair of processors; concurrent communication by several processors leads to serialization (Pattern 4 versus Patterns 5, 7, 8). The one-ported assumption seems reasonable: about the same bandwidth per processor is achieved when the processors within a node communicate with processors on different neighbors (Pattern 5 versus Patterns 7 and 8).

Patterns 5 and 6 hint at how the `nodeexchange` should be implemented. The best total bandwidth (by some processors terminating earlier) is achieved by the so-called duplex-implementation of Pattern 6. The processor at the right in each time-step sends and receives data, while at the left one processor sends data and one receives. Concretely, processor 0 performs the following 4 steps: (1) Send(1)Recv(4), (2) Send(2)Recv(1), (3) Send(3)Recv(2), (4) Send(4)Recv(3). Processors 1 to 3 performs a Recv(0) followed by a Send(0) operation, while

Table 1. Running times in milliseconds for first and last process to complete for the exchange patterns for 512KBytes sent per process

| Pattern | Time of first | Time of last | Average Time |
|-------------|---------------|--------------|--------------|
| 1 | 7.32 | 7.29 | 7.31 |
| 2 | 10.24 | 13.75 | 12.90 |
| 3 | 10.06 | 10.68 | 10.35 |
| 4 | 7.82 | 7.80 | 7.81 |
| 5 | 29.55 | 31.14 | 30.54 |
| 6 (serial) | 7.79 | 38.66 | 26.35 |
| 6 (simplex) | 10.19 | 40.77 | 28.54 |
| 6 (duplex) | 15.36 | 30.89 | 26.21 |
| 7 | 29.15 | 30.31 | 29.76 |
| 8 | 24.13 | 31.27 | 28.87 |

processor 4 initiates the exchange by a `Send(0)` followed by a `Recv(0)`. As can be seen, by this pattern exactly one message is sent from right to left and one message from left to right node in any given time instant. The achieved completion times are contrasted to a serial implementation (all processes perform a `SendRecv` operation), and a “simplex”-operation in which in each time step either a send or a receive operation is performed.

4 Implementing the Exchange

For the Giganet cluster the best total inter-node bandwidth was achieved with the duplex exchange pattern 6. Huse [5] also observes that contention-free communication is important for a cluster with SCI interconnect, and achieves it by a token system: a token is passed around within each node, and the process which has the token is allowed to send (or receive). By generalizing pattern 6, we can achieve this *without* the use of tokens.

When two nodes are paired, the `nodeexchange` has to ensure that all processes with `done` $\leq i < \text{min}$ on the smaller node (in the \prec ordering) have exchanged data with all processes $0 \leq j < \text{nodesize}$ on the other node. The general case of the token-free duplex-exchange algorithm is given in Figure 2. An example exchange is shown in Figure 3; for simplicity we have assumed that `done` is 0. The exchange is initiated by the two black processes with local index `min` – 1, termed the *initiators*. As can be seen, the exchange pattern is such that at each step there is exactly one bidirectional data exchange between the two nodes. Double-arrows between processes on nodes *A* and *B* indicate an exchange performed either by a send and a receive or a combined send-receive operation. Arrows are annotated with the steps at which a transfer from *A* to *B* happens, plus the step at which the transfer from *B* to *A* happens (ignoring the exchanges along the

| Process $i < \text{min} - 1$ on “small” node A | Process $i = \text{min} - 1$ Process $i \geq \text{min}$ initiator on node B only on “large” node B |
|---|--|
| 1. for <code>done</code> $\leq j < i$: <code>Recv(j)</code> , <code>Send(j)</code> | for $0 \leq i < \text{min}$: |
| 2. <code>Recv(min - 1)</code> (from initiator) | <code>Send(i)Recv(i)</code> |
| 3. <code>Send(min)Recv(i)</code> | |
| 4. for $\text{min} \leq j < \text{nodesize} - 1$: | for <code>done</code> $\leq j < \text{min}$: |
| <code>Send(j + 1)Recv(j)</code> | <code>Recv(j)</code> , <code>Send(j)</code> |
| 5. <code>Send(i)Recv(nodesize - 1)</code> | |
| 6. for $i < j < \text{min} - 1$: <code>Send(j)</code> , <code>Recv(j)</code> | |
| 7. <code>Send(min - 1)</code> (back to initiator) | |

Fig. 2. The token-free duplex-exchange algorithm for the case where node *B* has more processes than node *A*: initiators, small and large processes. `Send/Recv(i)` denotes a send/receive-operation to the process with local index *i* on the other node. If the nodes have the same size, steps 3-5 for node *A* are replaced by a single `SendRecv(i)` operation

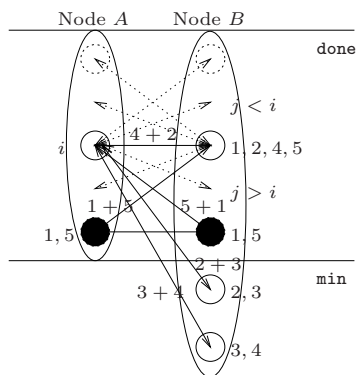


Fig. 3. Token free exchange between nodes

dotted lines, which are completely symmetric). Process i on the smaller node is active in all steps. The annotations at the other processes indicate the steps in which they are active. As can be seen, the initiators receive their data from the process with local rank i at the same time step. For the complete exchange, initiators have to iterate over all processes $i < \min - 1$ (see Figure 2). We have sketched the proof of the following proposition.

Proposition 1. *Assume that sending an amount of data from one node to another takes one unit of time. If the two initiators start at the same time sending their data to processes with local rank i , they will both receive data from processes i at the same time, and the processes with local rank i will have completed exchange with all processes at the opposite node. At each step of the exchange exactly one process at each node is sending data, and exactly one process at each node is receiving data.*

5 Experimental Evaluation

We finally present some results for hierarchical factor-algorithm and compare to non-hierarchical `MPI_Alltoall` implementations. The scheduler for our Giganet cluster allocates the processes in `MPI_COMM_WORLD` to the SMP nodes in a block-wise fashion. By splitting `MPI_COMM_WORLD` we can achieve various distributions of processes over the SMP nodes.

We report the average completion time for `MPI_Alltoall` for various process distributions. Results for 128Bytes and 512KBytes (per process) are shown in Table 2 for the the trivial algorithm, a *flat handshake algorithm* (in round i each process u performs a send-receive operation with processes $u - i \bmod n$ and $u + i \bmod n$), the *flat factor-algorithm*, and the *hierarchical factor-algorithm*.

The results are, for the Giganet cluster, somewhat disappointing. For the distributions reported here the hierarchical factor-algorithm is slightly better than the trivial algorithm, but only for data sizes larger than around

Table 2. Average completion times for 4 algorithms for `MPI_Alltoall` communication on the Giganet cluster with 6 nodes and 4 processors per node. Datasize is the amount of data sent to each other process

| Process distribution | Datasize | Trivial | Flat handshake | Flat factor | Hierarchical factor |
|----------------------|-----------|---------|----------------|-------------|---------------------|
| Full | 128Bytes | 0.76 | 1.26 | 1.10 | 1.84 |
| (4 processes/node) | 512KBytes | 666.70 | 654.77 | 710.01 | 589.26 |
| Sparse | 128Bytes | 0.08 | 0.15 | 0.15 | 0.15 |
| (1 process/node) | 512KBytes | 56.05 | 49.73 | 50.38 | 50.36 |
| Organ pipe | 128Bytes | 0.37 | 0.55 | 0.55 | 1.08 |
| (1,2,3,4,3,2) | 512KBytes | 326.75 | 313.90 | 343.87 | 302.78 |
| Random | 128Bytes | 0.66 | 0.82 | 0.82 | 1.67 |
| (21 processes) | 512KBytes | 535.12 | 542.37 | 576.43 | 520.73 |

16KBytes/process. The improvement in average completion time is about 10%. The sparse distribution shows that pairing, either by factoring or handshake makes sense for non-hierarchical systems. As should be, the hierarchical factor algorithm degenerates into the standard (flat) factor algorithm in this special case. For the other, non-sparse distributions it is interesting to note that the hierarchical factor algorithm performs considerably better than the flat factor algorithm. For small messages the hierarchical factor algorithm performs badly. For this case it can probably be improved by intra-node gather/scatter such that the exchange is done with only one communication operation. We have not yet tried this.

We have reasons to believe that the results are system dependent: (a) in [5] larger improvements are shown for an SCI-based cluster with only 2 processors per node, and (b) on a 4-node NEC SX-6 with 8 processors per node, improvements of 25% in completion times are observed, increasing to *factors of more than two* when going to a large system with crossbar interconnect. It would be interesting to experiment further on other SMP-systems with more processors (per node), and interconnects with other characteristics.

6 Concluding Remarks

We have described the implementation on a small Giganet SMP of a new algorithm for regular, personalized all-to-all communication which takes the hierarchical communication structure explicitly into account. Despite a careful implementation, in which in particular the implementation of the `nodeexchange` was quite crucial, only modest improvements over a completely trivial algorithm were achieved: all-to-all communication is a difficult problem. Also, the assumptions on which the algorithm was based of bidirectional, one-ported communication capabilities were only partially fulfilled by the system, and it was not known

how multiple simultaneous requests are handled by the (older) Linux kernel and the network card of our Giganet cluster.

Our algorithm is based on the assumption of homogeneous communication between SMP nodes. For large, heterogeneous SMP systems this may not be valid, and for such cases a modified algorithm is needed.

References

- [1] M. Golebiewski and J. L. Träff. MPI-2 one-sided communications on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, pages 16–23, 2001. 393, 395
- [2] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998. 392
- [3] S. E. Hambruch, F. Hameed, and A. A. Khokhar. Communication operations on coarse-grained mesh architectures. *Parallel Computing*, 21(5):731–752, 1995. 393
- [4] F. Harary. *Graph Theory*. Addison-Wesley, 1967. 393
- [5] L. P. Huse. MPI optimization for SMP based clusters interconnected with SCI. In *7th European PVM/MPI User's Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 56–63, 2000. 392, 397, 399
- [6] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 377–386, 2000. 392
- [7] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 492–499, 2000. 392
- [8] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaata, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, volume 34(8) of *ACM Sigplan Notices*, pages 131–140, 1999. 392
- [9] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaata, and R. A. F. Bhoedjang. MPI's reduction operations in clustered wide area systems. In *Third Message Passing Interface Developer's and User's Conference (MPIDC'99)*, pages 43–52, 1999. 392
- [10] P. Sanders and R. Solis-Oba. How helpers hasten h -relations. *Journal of Algorithms*, 41(1):86–98, 2001. 393
- [11] P. Sanders and J. L. Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 786–790, 2002. 392, 394, 395
- [12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998. 392
- [13] X. Wang, E. K. Blum, D. S. Parker, and D. Massey. The dance party problem and its application to collective communication in computer networks. *Parallel Computing*, 23(8):1141–1156, 1997. 393

Fujitsu MPI-2: Fast Locally, Reaching Globally

Georg Bißeling, Hans-Christian Hoppe, Alexander Supalov¹,
Pierre Lagier², and Jean Latour³

¹ Pallas GmbH, a member of ExperTeam Group
Hermülheimer Str. 10, D-50321 Brühl, Germany
`supalov@pallas.com`

² Fujitsu European Centre for Information Technology
8, rue Maryse Hilsz, F-31500 Toulouse, France
`lagier@fujitsu.fr`

³ Fujitsu Systems Europe, ORLYTECH
3, allée Hélène Boucher, F-91781 Wissous Cedex, France
`latour@fujitsu.fr`

Abstract. Fujitsu MPI-2 runs on a variety of platforms - from Fujitsu VPP vector machines through Sparc SMP computers to Linux SMP clusters - and brings to all of them a fully implemented MPI-2 together with the full IMPI functionality. The implementation happens to match or beat the respected competition on the intramachine benchmarks - and yet reach all over the world via TCP/IP networks.

1 Introduction

It is difficult enough to implement the whole set of the MPI standards and its relations [1, 3, 5]. Keeping the implementation in tune with the corrections and additions [2, 4, 6], ongoing discussions [7] and possible future extensions [8] is not all too easy either. But the real challenge lies in providing them all in one package, and still remaining fast on several target platforms and between them, and compatible with the rest of the world.

Fujitsu MPI-2 manages just that. It has been created in a long term project that involved tight co-operation between Pallas and Fujitsu. We do not have enough place here even to mention all of the remarkable personalities involved, and use this opportunity to thank them all for their dedicated and skillful efforts.

The text is organized as follows. Section 2 introduces the general design ideas behind the Fujitsu MPI-2 implementation. Section 3 goes into high level details of the software. Section 4 illustrates just how well Fujitsu MPI-2 performs on the target platforms by providing comparison with the directly competing software. Finally, Section 5 sums up the discussion and dwells upon the plans pursued by the joint Fujitsu MPI-2 development team.

2 General Design

Although by inspecting the code one may still vaguely infer that the Fujitsu MPI-2 started as a certain version of MPICH (like many other successful vendor

implementations, see <http://www-unix.mcs.anl.gov/mpi/mpich>), the specifics of the target platforms, the history of the project, and the challenges of the grid computing have imprinted themselves onto the current shape of the code.

Fujitsu MPI-2 has originally been targeted at the Fujitsu VPP vector multiprocessors with distributed memory, based on the proprietary 32- and 64-bit processors running under UXP/V operating system. Then it was extended to cover the symmetric multiprocessor configurations of the Fujitsu GP and PrimePower series running on Sparc compatible processors under control of the Solaris operating system. Of late, efforts are undertaken in porting the software to the clustered Linux SMP platforms based on the Intel processors.

As any reasonable software package, Fujitsu MPI-2 has a layered structure. The top level provides the user with the complete MPI-2 implementation [3] complemented by the IMPI capability for starting and running MPI-1 jobs that span several machines connected via TCP/IP. The supported external language interfaces include C, C++ (derived from the portable MPI C++ interface implementation of the University of Notre Dame du Lac, see <http://www.osl.iu.edu/research/mpi2c++/>), and Fortran (in its full variant).

The medium layer provides environment management, process control, and data transfer services to the upper layer. The data transfer includes the local point-to-point, collective, and one-sided communication. It is complemented by an efficient TCP/IP message passing engine and a flexible file I/O subsystem.

Parallel jobs started using Fujitsu MPI-2 can work in the so called limited and full modes. In the limited mode, all processes are equal and participate in the MPI job as peers - which is the traditional MPI approach. In the full mode, one more non-MPI process - called hereafter Fujitsu MPI-2 server process, or simply `mpiexec` - is started first as the father of all MPI processes that are created later by it.

The `mpiexec` process was initially used for controlling the MPI job and performing file I/O in case when the specially developed high performance MPI file system - called MPIFS - was deployed. Of late, providing connectivity to other `mpiexec` processes and the rest of the world via TCP/IP sockets has become another important task. Note that the `mpiexec` is not necessary when a Fujitsu MPI-2 job does not require TCP/IP connectivity or MPIFS (for the Unix and Network File Systems are supported in the limited mode as well).

3 Implementation Details

The combination of the very fast local communication with the much slower external network connections has always been a major challenge in the design of the message passing libraries. This section will show how Fujitsu MPI-2 manages to solve this intricate problem.

3.1 Local Data Transfer

Thanks to the efforts of the joint development team, the Fujitsu MPI-2 medium layer can always rely on the well defined and fast low level capabilities provided by the xMPLib libraries for every supported platform. Of them, the MPLib is used for interacting with the VPP computers; this library was developed by Fujitsu. In the SMP world, the experimental SMPLib, which is provided by Pallas, covers the pure SMP configurations (without vendor specific networks), while the GMPLib, which is being developed by Fujitsu, is intended to include the latest clustering hardware solutions from Fujitsu as well.

The recent introduction of process attachments effectively lead to a multidevice design instead of a monodevice design that had been adhered to so far. Most of the provisions for heterogeneity and multidevice support that were present in the MPICH heritage had been abandoned for efficiency reasons. This left us with the question how to remain as fast as before locally, and correct in terms of MPI progress and message ordering requirements.

The most difficult problem was that during receiving of messages using the `MPI_ANY_SOURCE` as the source process rank in a communicator that contained both local and remote processes, one had to listen to the fast connections with the local processes, and simultaneously ensure that the eventual messages from the remote processes are processed in due order.

The solution was to let the `mpiexec` appear as a local agent of the remote processes on the xMPLib level using the technique we called *xMPLib tunneling*. Essentially, it means that the `mpiexec` servers pass messages between remote processes so that the target process first receives a message from its `mpiexec` that carries the sender process rank and the proper tag, but contains no user data. Once such a message arrives, the `mpiexec` and the target process transfer the actual data using the xMPLib one-sided communication. To this end, the data is stored temporarily by the target server process, with due attention paid to the flow control in case of a threatening buffer overrun, of course.

Thanks to this method, a detailed discussion of which would unfortunately explode this article, the local data transfer that bypasses the `mpiexec` is as efficient as it was. All functions served by the `mpiexec` are also executed at the same speed as before, provided that no network functions are used. As soon as there are some network connections, one has however to pay a little extra in terms of latency, for instance, while performing the MPIFS file I/O via the `mpiexec`.

3.2 Process Attachment

When the most creative users got their hands on process attachments, they instantly realized that process spawning together with attachments offered them a whole new world of flexible, dynamically started subjobs that connect to each other as the control flow in the application requires. This resulted in data driven designs featuring a driver program that schedules subjobs as soon as all necessary data objects are ready (see http://www.cerfacs.fr/globc/PALM_WEB for

details). To support this kind of application a simple name service with intrajob scope was added to `mpiexec`.

Notice that in case of an intrajob process attachment the `mpiexec` acts as an TCP server and an TCP client at the same time. To prevent deadlocks in this situation one has to take care that all network related calls are to be carried out only when they can be done in a nonblocking fashion (like `connect()`), or when the data and buffer space are available (e.g., `read()`, `write()` and `accept()`).

The only drawback the application programmers now complained about was that the local TCP/IP connections were much slower (between one and two orders of magnitude both in latency and bandwidth) when compared to the native communication. This was observed despite the provision of smart TCP/IP shortcuts on the operating system level. In addition, it was possible to use neither the MPI-2 one-sided communication nor file I/O over the attached connections even though the processes actually ran inside one xMPLib universe.

So we came up with the so called *fast attachment* feature that uses the native high speed data connection whenever an intramachine TCP/IP connection is detected. This is entirely transparent to the applications programmer but configurable at runtime.

To enable fast attachments, one uses a command line like `mpiexec -fa MyJob prog`. The network connection is built via TCP/IP as usual but both ends negotiate to use fast attachments for the job `MyJob` so they exchange the local MPLib addresses of their participating processes and terminate the TCP/IP connection. Afterwards the new communicator returned by the `MPI_Comm_connect` and `MPI_Comm_accept` is built upon the local xMPLib addresses. This negotiation allows to keep slow attachments via TCP working for all connections that span more than one job.

The result of all this is a fast connection that performs *exactly as good* as a connection between processes that began their lives during the job startup, with the added ability to use the MPI-2 one-sided communication and file I/O over the communicator involved.

3.3 IMPI Capability

IMPI provides an environment for MPI-1 jobs that span several machines connected via TCP/IP and possibly running different vendor MPIS. For example, it should allow to connect a little endian Linux cluster running LAM MPI to a big endian Fujitsu VPP 5000.

When the TCP/IP engine was being designed for the integration of MPI-2 process attachments, the implementation of IMPI had already been put into the development schedule. So the design of the engine did incorporate hooks for the introduction of the elaborate and configurable IMPI flow control protocol and for the use of IPv6 network addresses. The same is true for the simple IMPI authentication mechanism.

It was decided to use the IMPI startup server developed by the University of Notre Dame du Lac (see <http://www.osl.iu.edu/research/impi>). We would like to especially thank Jeff Squyres for his support and fruitful email discussions.

Our first step was to provide an IMPI simulation that allowed us to run a single local job as if it were the only client in an IMPI job of its own. This enabled us to use our MPI-1 test suite to check the bigger part of the IMPI-required reimplementation of all collective operations without having the actual TCP/IP engine ready.

The next step was to implement the IMPI control flow protocol and to test it talking locally.

The final step – in fact a long march – was to test our implementation against the Java-based IMPI test suite provided by the National Institutes for Standards and Technology (NIST) (see <http://impi.nist.gov/IMPI>). We'd like to thank William George of NIST for his constant help and support.

4 Performance Results

The results presented in this section compare the performance of the current Fujitsu MPI-2 and the Sun HPC 4.0 software that we were able to get hold of for our comparative testing at the time when it was performed.

The comparison of the Fujitsu MPI-2 and Sun HPC was done on the same machine (for this test series we used the Sun Fire 6800 installation of the RWTH Aachen, Germany). It should be noted that the timings above have been obtained using an experimental native collective operation interface at the xMPLib level which we cannot dwell upon here for the lack of space.

We took the established benchmark suite PMB 2.2 used routinely by Pallas and many others for independent assessment of the quality of the MPI platforms (see <http://www.pallas.com/e/products/pmb>). The PMB suite exercises the most important MPI-2 communication modes and all application relevant collective operations.

In order to be able to compare the results in equal terms, we've decided to provide timings for the message sizes of 4 bytes and 4 Megabytes. The former size (4 bytes) looks strange at first (as latency is conventionally measured at zero size messages), but by taking 4 bytes as a basis we are able to, first, eliminate all possible shortcuts allowed for the collective operations by the MPI standard; and second, to make the reductions work on the most typical data units.

The latter size (4 Megabytes) is expected to exercise the slowest available message transfer protocol (if those are supported). This message size seems to suggest bandwidth instead of timing as the most natural measure, but in the case of collective operations (and some collective patterns implemented using the point-to-point functions) the bandwidth becomes very hard to define to universal satisfaction.

In order to equalize the disparity between the relative complexity of the benchmark operations involved, all the graphs below show timings, so that *smaller* means *better* (see Figures 1-8). Note that for more than 2 processors, the PingPong and PingPing tests still involve 2 processors (with the rest waiting on `MPI_Barrier`). Also, the Barrier test, unlike the rest of the operations involved, is

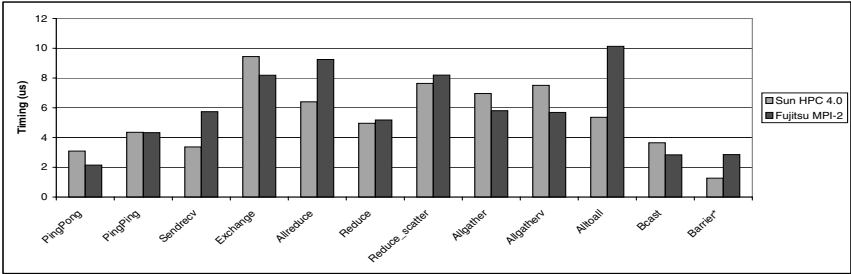


Fig. 1. PMB-MPI1 version 2.2 timings at 4 byte message size on 2 processors

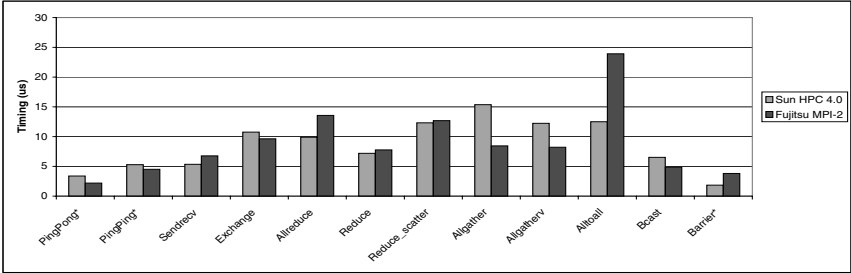


Fig. 2. PMB-MPI1 version 2.2 timings at 4 byte message size on 4 processors

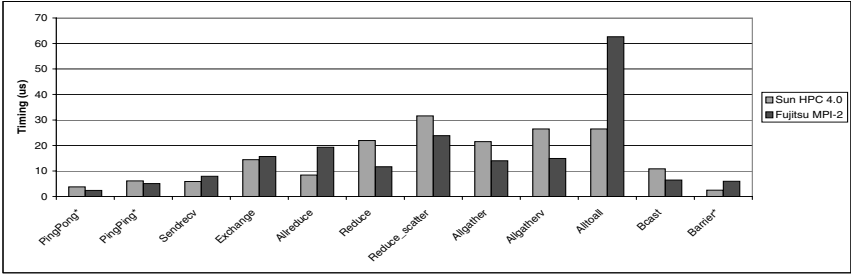


Fig. 3. PMB-MPI1 version 2.2 timings at 4 byte message size on 8 processors

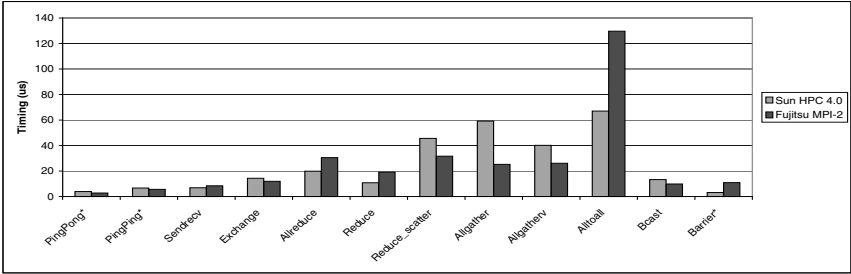


Fig. 4. PMB-MPI1 version 2.2 timings at 4 byte message size on 16 processors

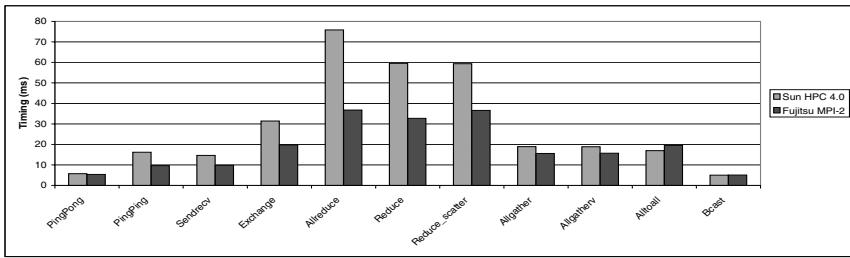


Fig. 5. PMB-MPI1 version 2.2 timings at 4 Megabyte message size on 2 processors

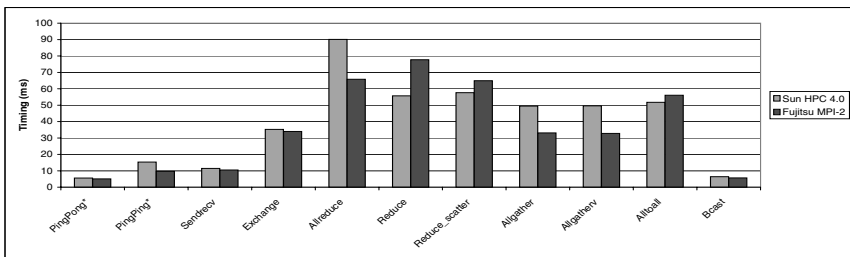


Fig. 6. PMB-MPI1 version 2.2 timings at 4 Megabyte message size on 4 processors

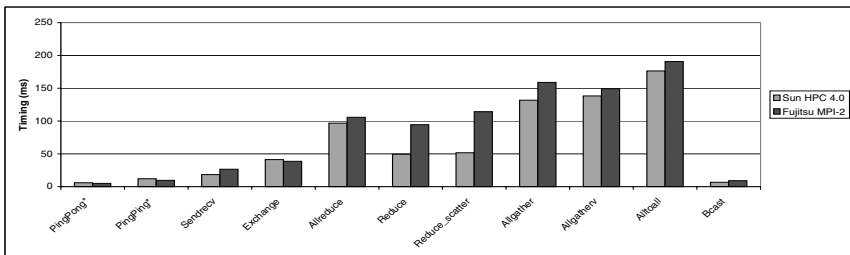


Fig. 7. PMB-MPI1 version 2.2 timings at 4 Megabyte message size on 8 processors

based on passing zero length messages or some other synchronization technique. This is why all these results are marked with asterisks.

Generally speaking, Fujitsu MPI-2 happens to match or beat Sun HPC on Sun's hardware with the notable exception of the Alltoall and Barrier tests. Also, although pairwise point-to-point communication of the Fujitsu MPI-2 appears superior, the Sun HPC collectives tend to scale better as of now. Further research and development that should eliminate the remaining disparities is currently underway.

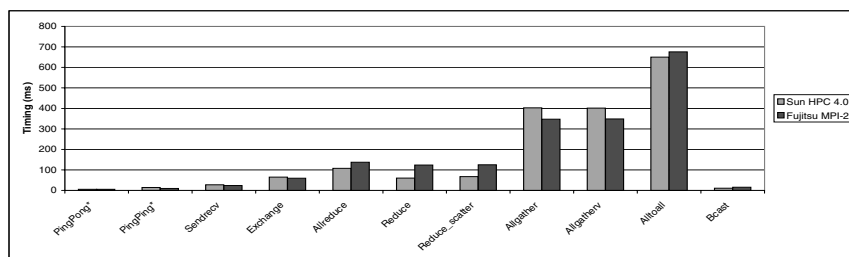


Fig. 8. PMB-MPI1 version 2.2 timings at 4 Megabyte message size on 16 processors

This is especially true of some reduction operations (like Allreduce), although on some operations the results appear quite equally balanced. It is interesting that currently in some cases, the message passing based collective operations may outperform their optimized versions, which fact is taken into account by using the optimizations adaptively only in those situations when they are beneficial.

5 Conclusions

This paper showed that it is possible to combine in one package the decent local performance (usually expected only from an MPI-1 implementation) with the global reach and versatility provided by the full blown MPI-2 and IMPI standards. It also showed the importance of a close co-operation between the upper MPI-2 layer and the underlying systems software.

The development teams at Fujitsu and Pallas continue to work together on the further improvement and extension of the Fujitsu MPI-2. Among the current plans is further improvement of the scalability of the collective operations and research on the role of firewalls on the grid communication. Also on our development agenda is the porting of the software to the clustered SMP machines running under Linux.

References

- [1] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. MPI Forum, June 12, 1995 401
- [2] Message Passing Interface Forum: Errata for MPI-1.1. MPI Forum, October 12, 1998 401
- [3] Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. MPI Forum, July 18, 1997 401, 402
- [4] Message Passing Interface Forum: Errata for MPI-2. MPI Forum, May 20, 1998 401
- [5] IMPI Steering Committee: IMPI: Interoperable Message-Passing Interface, Protocol Version 0.0. IMPI Steering Committee, January 2000 401

- [6] IMPI Steering Committee: Errata for IMPI version 0.0. IMPI Steering Committee, June 18, 2001 401
- [7] Message Passing Interface Forum: MPI-2.1 Discussion. Available at http://www.mpi-forum.org/mpi2_1 401
- [8] Message Passing Interface Forum: MPI-2: Journal of Development. MPI Forum, July 18, 1997 401

Communication and Optimization Aspects on Hybrid Architectures

Rolf Rabenseifner

High-Performance Computing-Center (HLRS), University of Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
rabenseifner@hlrs.de
www.hlrs.de/people/rabenseifner/

Abstract. Most HPC systems are clusters of shared memory nodes. Parallel programming must combine the distributed memory parallelization on the node inter-connect with the shared memory parallelization inside of each node. The hybrid MPI+OpenMP programming model is compared with pure MPI and compiler based parallelization. The paper focuses on bandwidth and latency aspects, but also whether programming paradigms can separate the optimization of communication and computation. Benchmark results are presented for hybrid and pure MPI communication.

Keywords: OpenMP, MPI, Hybrid Parallel Programming, Threads and MPI, HPC.

1 Motivation

The hybrid MPI+OpenMP programming model on clusters of SMP nodes is already used in many applications, but often there is only a small benefit as, e.g., reported with the climate model calculations of one of the Gordon Bell Prize finalists at SC 2001 [6], or sometimes losses are reported compared to the pure MPI model, e.g., as shown with an discrete element modeling algorithm in [4]. In the hybrid model, each SMP node is executing one multi-threaded MPI process. With pure MPI programming, each processor executes a single-threaded MPI process, i.e., the cluster of SMP nodes is treated as a large MPP (massively parallel processing) system.

One of the major drawbacks of the hybrid MPI-OpenMP programming model is based on a very simple usage of this hybrid approach: If the MPI routines are invoked only outside of parallel regions, all threads except the master thread are sleeping while the MPI routines are executed.

This paper will discuss this phenomenon and other hybrid MPI-OpenMP programming strategies. Sect. 2 shows different methods to combine MPI and OpenMP. Further rules on hybrid programming are discussed in Sect. 3, and pure MPI on hybrid architectures in Sect. 4. Sect. 5 presents benchmark results of the communication in both models. Sect. 6 to 8 compare the MPI based programming models with compiler based parallelization.

2 MPI and Thread-Based Parallelization

The combination of MPI and thread-based parallelization was already addressed by the MPI-2 Forum in Sect. 8.7 *MPI and Threads* in [8]. For hybrid programming, the MPI-1 routine `MPI_Init()` should be substituted by a call to `MPI_Init_threads()` which has the input argument named *required* to define which thread-support the application requests from the MPI library, and the output argument *provided* which is used by the MPI library to tell the application which thread-support is available. MPI libraries may support the following thread-categories (higher categories are supersets of all lower ones):

MPI_THREAD_SINGLE – No thread-support.

MPI_THREAD_FUNNELED – Only the master thread is allowed to call MPI routines. The other threads may run other application code while the master thread calls an MPI routine.

MPI_THREAD_SERIALIZED – Multiple threads may make MPI-calls, but only one thread may execute an MPI routine at a time.

MPI_THREAD_MULTIPLE – Multiple threads may call MPI without any restrictions.

The constants `MPI_THREAD_...` are monotonically increasing.

Between `MPI_THREAD_SINGLE` and `FUNNELED`, there are intermediate levels of thread support, not yet addressed by the standard:

T1a – The MPI process may be multi-threaded but only the master thread may call MPI routines **AND** only while the other threads do not exist, i.e., parallel threads created by a parallel region must be destroyed before an MPI routine is called. An MPI library supporting this class (and not more) must also return `provided=MPI_THREAD_SINGLE` (i.e., no thread-support) because of the lack of this definition in the MPI-2 standard¹.

T1b – The definition T1a is relaxed in the sense that more than one thread may exist during the call of MPI routines, but all threads except the master thread must sleep, i.e., must be blocked in some OpenMP synchronization. As in T1a, an MPI library supporting T1b but not more must also return `provided=MPI_THREAD_SINGLE`.

Usually, the application cannot distinguish whether an OpenMP based parallelization or an automatic parallelization needs T1a or T1b to allow calls to MPI routines outside of OpenMP parallel regions, because it is not defined, whether at the end of a parallel region the team of threads is sleeping or is destroyed. And usually, this category is chosen, when the MPI routines are called outside of parallel regions. Therefore, one should summarize the cases T1a and T1b to only one case:

T1 – The MPI process may be multi-threaded but only the master thread may call MPI routines **AND** only outside of parallel regions (in case of OpenMP) or outside of parallelized code (if automatic parallelization is used). We define here an additional constant **THREAD_MASTERONLY** with a value between `MPI_THREAD_SINGLE` and `MPI_THREAD_FUNNELED`.

¹ This may be solved in the revision 2.1 of the MPI standard.

3 Rules with Hybrid Programming

THREAD_MASTERONLY defines the most simple hybrid programming model with MPI and OpenMP, because MPI routines may be called only outside of parallel regions. The new cache coherence rules in OpenMP 2.0 guarantee that the outcome of an MPI routine is visible to all threads in a subsequent parallel region, and that the outcome of all threads of a parallel region is visible to a subsequent MPI routine.

The programming model behind MPI_THREAD_FUNNELED can be achieved by surrounding the call to the MPI routine with the OMP MASTER and OMP END MASTER directives inside of a parallel region. One must be very careful, because OMP MASTER does not imply an automatic barrier synchronization or an automatic cache flush neither at the entry to nor at the exit from the master section. If the application wants to send data computed in the previous parallel region or wants to receive data into a buffer that was also used in the previous parallel region (e.g., to use the data received in the previous iteration), then a barrier with implied cache flush is necessary prior to calling the MPI routine, i.e., prior to the master section. If the data or buffer is also used in the parallel region after the exit of the MPI routine and its master section, then also a barrier is necessary after the exit of the master section. If both barriers must be done, then while the master thread is executing the MPI routine, all other threads are sleeping, i.e., we are going back to the case T1b.

The rules of MPI_THREAD_SERIALIZED can be achieved by using the OMP SINGLE directive, which has an implied barrier only at the exit (unless NOWAIT is specified). Here again, the same problems as with FUNNELED must be taken into account.

These problems with FUNNELED and SERIALIZED arise, because the communication must be funneled from all threads to one thread (an arbitrary thread with OMP SINGLE, and the master thread with OMP MASTER). Only MPI_THREAD_MULTIPLE allows a direct message passing from each thread in one node to each thread in another node.

Based on these reasons and because THREAD_MASTERONLY is available on nearly all clusters, often, hybrid and portable parallelization is using only this parallelization scheme. This paper will evaluate this hybrid model by comparing it with the non-hybrid pure MPI model described in the next section.

4 Pure MPI on Hybrid Architectures

Using a pure MPI model, the cluster must be viewed as a hybrid communication network with typically fast communication paths inside of each SMP node and slower paths between the nodes. It is important to implement a good mapping of the communication paths used by application to the hybrid communication network of the cluster. The MPI standard defines virtual topologies for this purpose, but the optimization algorithm isn't yet implemented in most MPI implementations. Therefore, in most cases, it is important to choose a good

ranking in `MPI_COMM_WORLD`. E.g., on a Hitachi SR8000, the MPI library allows two different ranking schemes, round robin (ranks 0, N , $2*N$, ... on node 0; ranks 1, $N+1$, $2*N+1$, ... on node 1, ...; with N =number of nodes) and sequential (rank 0–7 on node 0, ranks 8–15 on node 1, ...), and the user has to decide which scheme may fit better to the communication needs of his application.

The pure MPI programming model implies additional message transfers due to the higher number of MPI processes and higher number of boundaries. Let us consider, for example, a 3-dimensional cartesian domain decomposition. Each domain may have to transfer boundary information to its neighbors in all six cartesian directions ($\updownarrow \rightleftharpoons \swarrow \nearrow$). Bringing this model on a cluster with 8-way SMP nodes, on each node, we should execute the domains belonging to a $2 \times 2 \times 2$ cube. Domain-to-domain communication occurs as node-to-node (inter-node) communication and as intra-node communication between the domains inside of each cube. Hereby, each domain has 3 neighbors inside the cube and 3 neighbors outside, i.e., in the inter-node and the intra-node communication the amount of transferred bytes should be equivalent. If we compare this pure MPI model with a hybrid model, assuming that the domains (in the pure MPI model) in each $2 \times 2 \times 2$ cube are combined to a super-domain in the hybrid model, then the amount of data transferred on the node-interconnect should be the same in both models. This implies that in the pure MPI model, the total amount of transferred bytes (inter-node plus intra-node) will be twice the number of bytes in the hybrid model. The same ratio is shown in the topology in the left diagram of Fig. 1. In the symmetric case, the intra-node and inter-node communication has the same transfer volume.

5 Benchmark Results

The following benchmark results will compare the communication behavior of the hybrid MPI+OpenMP model with the pure MPI model that can be named also as MPP-MPI model. Based on the domain decomposition scenario discussed in the last section, we compare the bandwidth of both models and the ratio of the total communication time presuming that in the pure MPI model, the total amount of transferred data is twice the amount in the hybrid model. The benchmark was done on a Hitachi SR8000 with 16 nodes from which 12 nodes are available for MPI parallel applications. Each node has 8 CPUs. The effective communication benchmark `b_eff` is used [5, 12]. It accumulates the communication bandwidth values of the communication done by each MPI process. To determine the bandwidth of each process, the maximum time needed by all processes is used, i.e., this benchmark models an application behavior, where the node with the slowest communication controls the real execution time. To compare both models, we use the following benchmark patterns:

- `b_eff` – the accumulated bandwidth average for several ring and random patterns (this is the major benchmark pattern of the `b_eff` benchmark);

Table 1. Comparing the hybrid and the MPP communication needs

| | | b _{eff} (avg.) | b _{eff} at Lmax | 3D-cyclic (average) | 3D-cyclic at Lmax |
|----------------------|--------------|----------------------------|-----------------------------|------------------------|----------------------|
| b_{hybrid} | [MB/s] | 1535 | 5565 | 1604 | 5638 |
| (per node) | [MB/s] | (128) | (464) | (134) | (470) |
| b_{MPP} | [MB/s] | 5299 | 16624 | 5000 | 18458 |
| (per process) | [MB/s] | (55) | (173) | (52) | (192) |
| b_{MPP}/b_{hybrid} | (measured) | 3.45 | 2.99 | 3.12 | 3.27 |
| s_{MPP}/s_{hybrid} | (assumed) | 2 | 2 | 2 | 2 |
| T_{hybrid}/T_{MPP} | (concluding) | 1.73 | 1.49 | 1.56 | 1.64 |

- 3D-cyclic – a 3-dimensional cyclic communication pattern with 6 neighbors for each MPI process (this is an additional pattern measured by the b_{eff} benchmark);

With the following sub-options, we get 4 metrics (columns) in Table 1:

- average – the average bandwidth of 21 different message sizes (8 byte – 8 MB);
- at Lmax – the bandwidth is measured with 8 MB messages.

For each metrics, the following rows are presented in Tab. 1:

- b_{hybrid} , the accumulated bandwidth b for the hybrid model measured with a 1-threaded MPI process on each node (12 MPI processes),
- and in parentheses the same bandwidth per node,
- b_{MPP} , the accumulated bandwidth for the pure MPI model (96 MPI processes with sequential ranking in MPI_COMM_WORLD),
- and in parentheses the same bandwidth per process,
- b_{MPP}/b_{hybrid} , the ratio of accumulated MPP bandwidth and accumulated hybrid bandwidth,
- T_{hybrid}/T_{MPP} , the ratio of execution times T , assuming that total size s of the transferred data in the pure MPI model is twice of the size in the hybrid model, i.e., $s_{MPP}/s_{hybrid} = 2$, as shown in Sect. 4. For this calculation, it is assumed that the measured bandwidth values are approximately valid also for doubled message sizes.

Note that this comparison was done with no special optimized topology mapping in the pure MPI model. The result shows that the pure MPI communication model is faster than the communication in the hybrid model. There are at least two reasons: (1) In the hybrid model, all communication was done by the master thread while the other threads were inactive; (2) One thread is not able to saturate the total inter-node bandwidth that is available for each node.

Fig. 1 shows a similar experiment. In the hybrid MPI+OpenMP communication scheme, only the left thread sends inter-node messages. Therefore, the

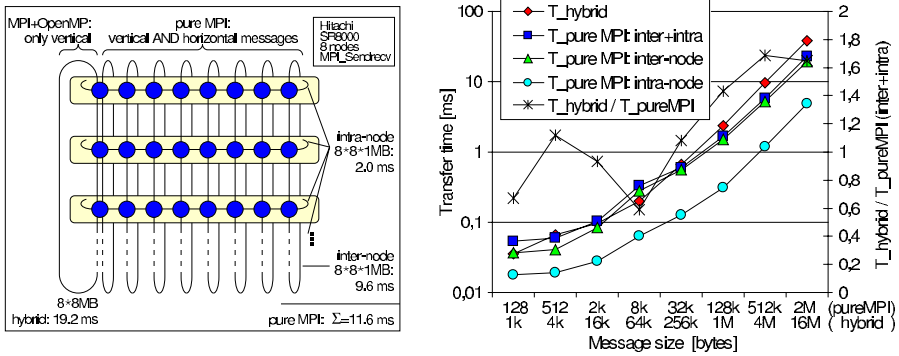


Fig. 1. Parallel communication in a cartesian topology

message size is 8 times the size used in the pure MPI scheme. Here, each CPU communicates in the vertical (inter-node) and horizontal (intra-node) direction. The total communication time with the hybrid model (19.2ms) is 66% greater than with the pure MPI communication (11.6ms), although with pure MPI, the total amount of transferred data is doubled due to the additional intra-node communication. The left diagram shows the measured transfer time for several message sizes and the ratio of the transfer time in the hybrid model to the transfer time of inter-node plus intra-node communication. Note that for the hybrid measurements, the message size must reflect that the inter-node data exchange of all threads is communicated by the master thread, and therefore, the message size is chosen 8 times larger, i.e., it ranges from 1 kB to 16 GB. The diagram shows that for message sizes greater than 32 kB, the pure MPI model is faster than the hybrid model in this experiment. With smaller message sizes, the ratio $T_{\text{hybrid}}/T_{\text{pureMPI}}$ depends mainly on the latencies of the underlying protocols that may differ due to the larger message sizes in the hybrid model.

A similar communication behavior can be expected on other platforms if the inter-mode network cannot be saturated by a single processor in each SMP node. This paper cannot analyze the reasons based on decisions in the hardware or software (MPI library) design of a system. For example, additional local copying for user space to system space and bad pipelining or parallelization of process-local activities and inter-node data transfer may cause that one CPU cannot reach the inter-node peak bandwidth. The shown ratio of hybrid to pure MPI transfer time may be a major reason when an application is running faster in the pure MPI model than in the hybrid model.

6 Comparison of Hybrid MPI+OpenMP versus Pure MPI

The comparison in this paper focuses on bandwidth and latency aspects, i.e., how to achieve a major percentage of the physical inter-node network bandwidth with various parallel programming models.

Although the benchmark results in the last section show advantages of the pure MPI model, there are also advantages of the hybrid model. In the hybrid model there is no communication overhead inside of a node. The message size of the boundary information of one process may be larger (although the total amount of communication data is reduced). This reduces latency based overheads. The number of MPI processes is reduced. This may cause a better speedup based on Amdahl's law and may cause a faster convergence if, e.g., the parallel implementation of a multigrid numeric is only computed on a partial grid. To reduce the MPI overhead by communicating only through one thread, the MPI communication routines should be relieved by unnecessary local work, e.g., concatenation of data should be better done by copying the data to a scratch buffer with a thread-parallelized loop, instead of using derived MPI datatypes. MPI reduction operations can be split into the inter-node communication part and the local reduction part by using user-defined operations, but a local thread-based parallelization of these operations may cause problems because these threads are running while an MPI routine may communicate.

Hybrid programming is often done in two different ways: (a) the domain decomposition is used for the inter-node parallelization with MPI and also for the intra-node parallelization with OpenMP, i.e., in both cases, a coarse grained parallelization is used. (b) The intra-node parallelization is implemented as a fine grained parallelization, e.g., mainly as loop parallelization. The second case also allows automatic intra-node parallelization by the compiler, but Amdahl's law must be considered independently for both parallelizations.

Now we want to compare three different hybrid programming schemes: In the *masteronly* scheme, only the master thread communicates and only outside of parallel regions. The computation is parallelized on all CPUs of an SMP node and inside of parallel regions. In the *funneled* scheme, the communication on the master thread is done in parallel with the computation on the other threads. For this, the application has to be restructured to allow the overlap of communication and computation. In the *multiple* scheme, all threads may communicate and compute in parallel. If the other application threads do not sleep while the master thread is communicating with MPI then communication time T_{hybrid} in Tab. 1 counts only the eighth (a node has 8 CPUs on the SR8000) because only one instead of 1 (active) plus 7 (idling) CPUs is communicating. In this hybrid programming style, the factor T_{hybrid}/T_{MPP} must be reduced to the eighth, i.e. from about 1.6 to about 0.2. This can be implemented by dedicating one thread for communication and the other threads of a node for computing, but also with full load balancing with different mixes of computation and communication on all threads.

Wellein et al. compared in [14] the two hybrid programming schemes *masteronly* (named vector-mode in [14]) and *funneled* (task-mode). They show that the performance ratio $\epsilon = (\frac{T_{funneled \text{ or } multiple}}{T_{masteronly}})^{-1}$ of *funneled* (or *multiple*) to *masteronly* execution has the bounds $1 - \frac{1}{n} \leq \epsilon \leq 2 - \frac{1}{n}$ if n is the number of threads of each SMP node. In general, if m threads are reserved for communication, T_{COMM} and T_{COMP} being the accumulated communication and computation time, and $f := \frac{T_{COMM}}{T_{COMM} + T_{COMP}}$ being the real communication percentage, then ϵ is bounded² by $1 - \frac{m}{n} \leq \epsilon \leq 1 + m(1 - \frac{1}{n})$ and $\epsilon \leq 1 + fn(1 - \frac{1}{n})$. If $m \geq 1$ or $\frac{m}{n} \geq f$, then the *funneled* scheme is faster if $f \geq \frac{2}{n(n/m-1)+1}$. The maximum of ϵ is given for $f = \frac{m}{n}$. E.g., if $n=8$ and $m=1$, the first upper bound of ϵ indicates that the *funneled* scheme may be up to 1.875 times faster than *masteronly*, but if the communication ratio f does not fit to m/n , only small profits may be shown, as indicated with the second upper bound $1 + fn(1 - \frac{1}{n})$ and benchmarks in [14].

7 MPI versus Compiler-Based Parallelization

Now, we compare the MPI based models with the NUMA or RDMA based models. To access data on another node with MPI, the data must be copied to a local memory location (so called halo or shadow) by message passing, before it can be loaded into the CPU. Usually all necessary data should be transferred in one large message instead of using several short messages. Then, the transfer speed is dominated by the asymptotic bandwidth of the network, e.g., as reported for 3D-cyclic-Lmax in Tab. 1 per node (470 MB/s) or per process (192 MB/s). With NUMA or RDMA, the data can be loaded directly from the remote memory location into the CPU. This may imply short accesses, i.e., the access is latency bound. Although the NUMA or RDMA latency is usually 10 times shorter than the message passing latency, the total transfer speed may be worse. E.g., [2] reports on a ccNUMA system a latency of 0.33–1 μ s, which implies a bandwidth of only 8–24 MB/s for a 8 byte data. This effect can be eliminated if the compiler has implemented a remote pre-fetching strategy as described in [9], but this method is still not used in all compilers.

The remote memory access can also be optimized by buffering or pipelining the data that must be transferred. This approach may be hard to automate, and current research in OpenMP compiler technology already studies the bandwidth optimization on SMP clusters [13], but it can be easily implemented as an directive-based optimization technique: The application thread can define the (remote) data it will use in the next simulation step and the compiled OpenMP code can pre-fetch the whole remote part of the data with a bandwidth-optimized transfer method. Table 2 summarizes this comparison.

² If m is non-integer, $m < 1$, and $\frac{m}{n} < f$, then the lower bound is $m - \frac{m}{n} \leq \epsilon$.

Table 2. Memory copies from remote memory to local CPU register

| Access method | copies | remarks | bandwidth $b(\text{message size})$ |
|---|--------|---|--|
| 2-sided MPI | 2 | internal MPI buffer + application receive buffer | $b_{\infty} / (1 + \frac{b_{\infty} T_{lat}}{\text{size}})$, e.g., $300 \text{ MB/s} / (1 + \frac{300 \text{ MB/s} \times 10 \mu\text{s}}{10 \text{ kB}})$ $= 232 \text{ MB/s}$ |
| 1-sided MPI | 1 | application receive buffer | same formula, but probably better b_{∞} and T_{lat} |
| Co-Array Fortran, UPC, HPC, OpenMP with cluster extensions | 1 | page based transfer | extremely poor, if only parts of the page are needed |
| | 0 | word based access | 8 byte / T_{lat} , e.g., 8 byte / $0.33 \mu\text{s} = \mathbf{24 \text{ MB/s}}$ |
| | 0 | latency hiding with pre-fetch | b_{∞} |
| | 1 | latency hiding with buffering | see 1-sided communication |

8 Parallelization and Compilation

Major advantages of OpenMP based programming are that the application can be *incrementally parallelized* and that one still has a single source for serial and parallel compilation. On a cluster of SMPs, the major disadvantages are that OpenMP has a flat memory model and that it does not know buffered transfers to reach the asymptotic network bandwidth. But, these problems can be solved by tiny additional directives, like the proposed migration and memory-pinning directives in [3], and additional directives that allow a contiguous transfer of the whole boundary information between each simulation step. Those directives are optimization features that do not modify the basic OpenMP model, as this would be done with directives to define a full HPF-like user-directed data distribution (as in [3, 7]). Another lack in the current OpenMP standard is the absence of a strategy of combining automatic parallelization with OpenMP parallelization, although this is implemented in a non-standardized way in nearly all OpenMP compilers. This problem can be solved, e.g., by adding directives to define scopes where the compiler is allowed to automatically parallelize the code, e.g., with *auto-parallel* regions. An OpenMP-based parallel programming model for SMP-clusters should be usable for both, fine grained loop parallelization, and coarse grained domain decomposition. There should be a clear path from MPI to such an OpenMP cluster programming model with a performance that should not be worse than with pure MPI or hybrid MPI+OpenMP.

It is also important to have a good compilation strategy that allows the development of well optimizing compilers on any combination of processor, memory access, and network hardware. The MPI based approaches, especially the hybrid MPI+OpenMP approach, clearly separate remote from local memory access optimization. The remote access is optimized by the MPI library, and the local memory access must be improved by the compiler. Such separation is realized, e.g., in the NANOS project OpenMP compiler [1, 10]. The separation of local and remote access optimization may be more essential than the chance of achiev-

ing a zero-latency by remote pre-fetching (Tab. 2) with direct compiler generated instructions for remote data access. Pre-fetching can also be done via macros or library calls in the input for the local (OpenMP) compiler.

9 Conclusion

For many parallel applications on hybrid systems, it is important to achieve a high communication bandwidth between the processes on the node-to-node inter-connect. On such architectures, the standard programming models of SMP or MPP systems do not longer fit well. The rules for hybrid MPI+OpenMP programming and the benchmark results in this paper show that a hybrid approach is not automatically the best solution if the communication is funneled by the master thread and long message sizes can be used. The MPI based parallel programming models are still the major paradigm on HPC platforms. OpenMP with further optimization features for clusters of SMPs and bandwidth based data transfer on the node interconnect have a chance to achieve a similar performance together with an incremental parallelization approach, but only if the current SMP model is enhanced by features that allow an optimization of the total inter-node traffic. Same important is a strategy that allows independently the optimization of the computation (e.g., choosing the best available compiler for the processor and programming language) and the communication.

Acknowledgments

The author would like to acknowledge his colleagues and all the people that supported these projects with suggestions and helpful discussions. He would especially like to thank Alice Koniges, David Eder and Matthias Brehm for productive discussions of the limits of hybrid programming, Bob Ciotti and Gabrielle Jost for the discussions on MLP, Gerrit Schulz for his work on the benchmarks, Gerhard Wellein for discussions on network congestion in the pure MPI model, and Thomas Bönisch, Matthias Müller, Uwe Küster, and John M. Levesque for discussions on OpenMP cluster extensions and vectorization.

References

- [1] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver, *NanosCompiler: A Research Platform for OpenMP Extensions*, in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999. 418
- [2] Robert B. Ciotti, James R. Taft, and Jens Petersohn, *Early Experiences with the 512 Processor Single System Image Origin2000*, proceedings of the 42nd International Cray User Group Conference, SUMMIT 2000, Noordwijk, The Netherlands, May 22–26, 2000, www.cug.org. 417
- [3] Jonathan Harris, *Extending OpenMP for NUMA Architectures*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000. 418

- [4] D.S. Henty, *Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling*, in Proc. Supercomputing'00, Dallas, TX, 2000. <http://citeseer.nj.nec.com/henty00performance.html>. 410
- [5] Alice E. Koniges, Rolf Rabenseifner, Karl Solchenbach, *Benchmark Design for Characterization of Balanced High-Performance Architectures*, in proceedings, 15th International Parallel and Distributed Processing Symposium (IPDPS'01), Workshop on Massively Parallel Processing, April 23-27, 2001, San Francisco, USA. 413
- [6] Richard D. Loft, Stephen J. Thomas, and John M. Dennis, *Terascale spectral element dynamical core for atmospheric general circulation models*, in proceedings, SC 2001, Nov. 2001, Denver, USA. 410
- [7] John Merlin, *Distributed OpenMP: Extensions to OpenMP for SMP Clusters*, in proceedings of the Second European Workshop on OpenMP, EWOMP 2000. 418
- [8] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997, www.mpi-forum.org. 411
- [9] Matthias M. Müller, *Compiler-Generated Vector-based Prefetching on Architectures with Distributed Memory*, in High Performance Computing in Science and Engineering '01, W. Jäger and E. Krause (eds), Springer, 2001. 417
- [10] The NANOS Project, Jesus Labarta, et al., [//research.ac.upc.es/hpc/nanos/](http://research.ac.upc.es/hpc/nanos/). 418
- [11] OpenMP Group, www.openmp.org.
- [12] Rolf Rabenseifner and Alice E. Koniges, *Effective Communication and File-I/O Bandwidth Benchmarks*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 8th European PVM/MPI Users' Group Meeting, Santorini, Greece, LNCS 2131, Y. Cotronis, J. Dongarra (Eds.), Springer, 2001, pp 24-35, 413
www.hlrs.de/mpi/b_eff/, www.hlrs.de/mpi/b_eff_io/.
- [13] Mitsuhsa Sato, Shigehisa Satoh, Kazuhiro Kusano and Yoshio Tanaka, *Design of OpenMP Compiler for an SMP Cluster*, in proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden, Sep. 1999, pp 32-39. <http://citeseer.nj.nec.com/sato99design.html>. 417
- [14] G. Wellein, G. Hager, A. Basermann, and H. Fehske, *Fast sparse matrix-vector multiplication for TeraFlop/s computers*, in proceedings of Vector and Parallel Processing - VECPAR'2002, Porto, Portugal, June 26-28, 2002, Springer LNCS. 417

Performance Analysis for MPI Applications with SCALEA^{*}

Hong-Linh Truong¹, Thomas Fahringer¹,
Michael Geissler², and Georg Madsen³

¹ Institute for Software Science, University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
`{truong,tf}@par.univie.ac.at`

² Photonics Institute, Technical University Vienna
Gusshausstrasse 27/387, A-1040 Vienna, Austria
`geissler@tuwien.ac.at`

³ Institute of Physical and Theoretical Chemistry, Technical University Vienna
Getreidemarkt 9/156, A-1060 Vienna, Austria
`gmadsen@theochem.tuwien.ac.at`

Abstract. The performance of message passing programs can be challenging to comprehend. In previous work we have introduced SCALEA, which is a performance instrumentation, measurement, analysis, and visualization tool for parallel and distributed programs. In this paper we report on experiences with SCALEA for performance analysis of two realistic MPI codes taken from laser physics and material science. SCALEA has been used to automatically instrument - based on user provided directives - the source codes, to compute performance overheads, to relate them to the source code, and to provide a range of performance diagrams in order to explain performance problems as part of a graphical user interface. Multiple-experiment performance analysis allows to compare and to evaluate the performance outcome of several experiments which have been conducted on a SMP cluster architecture.

1 Introduction

Message passing serves as an effective programming technique for parallel architectures and also enables the exploitation of coarse-grain parallelism on distributed computers as evidenced by the popularity of the Message Passing Interface (MPI). The performance of parallel and distributed message passing programs can be difficult to understand. In order to achieve reasonable performance the programmer must be intimately familiar with many aspects of the application design, software environment, and the target architecture. Performance tools play a crucial role to support performance tuning of distributed and parallel applications.

^{*} This research is partially supported by the Austrian Science Fund as part of the Aurora Project under contract SFBF1104.

We have developed SCALEA [11, 10] which is a performance instrumentation, measurement, analysis, and visualization tool for parallel and distributed programs that focuses on post-mortem and on-line performance analysis for Fortran MPI, OpenMP, HPF, and mixed parallel programs. The approach supported by SCALEA is that it seeks to explain the performance behavior of each program by computing a variety of performance metrics based on a novel overhead classification [11, 10] for MPI, OpenMP, HPF and mixed parallel programs. In order to determine overheads, SCALEA divides the program sources into code regions and locates whether the performance problems occur in those regions or not. A highly flexible instrumentation and measurement system is provided which can be controlled by program directives and command line options.

A data repository is employed in order to store performance data (raw performance data and performance metrics) and information about performance experiments (e.g. source codes, machine configuration, etc.) which alleviates the association of performance information with experiments and the source code. SCALEA also supports multiple-experiment performance analysis that allows to compare and to evaluate the performance outcome of several experiments. A graphical user interface is provided to view the performance at the level of arbitrary code regions, processes and threads for single- and multi-experiments.

The rest of this paper is organized as follows: Related work is outlined in Section 2. Section 3 presents an overview of SCALEA. Experiments that demonstrate the usefulness of SCALEA are shown in Section 4, followed by conclusions and future work in Section 5.

2 Related Work

Paradyn [7] uses dynamic instrumentation and searches for performance bottlenecks based on a specification language. Performance analysis is done for functions and function calls. SCALEA has a more flexible mechanism to control the code regions for which performance metrics should be determined.

The Pablo toolkit[9] uses event tracing to develop performance tools for both message passing and data parallel programs. It provides an instrumentation interface that inserts user-specified instrumentation in the program. SCALEA is more flexible to control what performance metrics should be determined for which given code regions.

TAU [6] is a performance framework for instrumentation, measurement, and analysis of parallel Fortran and C/C++ programs. TAU supports multiple timing and hardware metrics. However, TAU does not provide a similar flexible mechanism as SCALEA to control instrumentation. SCALEA also provides more high-level performance metrics (e.g. data movement and control of parallelism overhead) than TAU.

Paraver [12] and VAMPIR [8] are performance analysis tools that also cover MPI programs. They process trace files to compute various performance results and provide a rich set of performance displays. SCALEA offers a larger set of

performance metrics and enables a more flexible mechanism to control instrumentation and measurements.

EXPERT [3] detects a range of performance properties for message passing, OpenMP, and mixed programs. SCALEA provides a more flexible mechanism to control instrumentation and measurement.

None of the above mentioned tools employ a data repository to organize and store performance data based on which higher-level performance analysis can be conducted. Moreover, these tools lack the support of multiple performance experiment analysis and a flexible mechanism to control instrumentation (except Paradyn) as provided by SCALEA.

3 SCALEA Overview

In this section we give a brief overview of SCALEA. For more details the reader may refer to [11, 10]. Figure 1 shows the architecture of SCALEA which consists of several modules: SCALEA Instrumentation System, SCALEA Runtime System, SCALEA Performance Analysis & Visualization System, and SCALEA Performance Data Repository.

The SCALEA Instrumentation System(SIS) supports automatic instrumentation of Fortran MPI, OpenMP, HPF, and mixed OpenMP/MPI programs. SIS enables the user to select (by directives or command-line options) code regions and performance metrics of interest.

Moreover, SCALEA offers an interface for other tools to traverse and annotate the AST to specify code regions for which performance metrics should be obtained. Based on pre-selected code regions and/or performance metrics, SIS automatically analyzes source codes and inserts probes (instrumentation code)

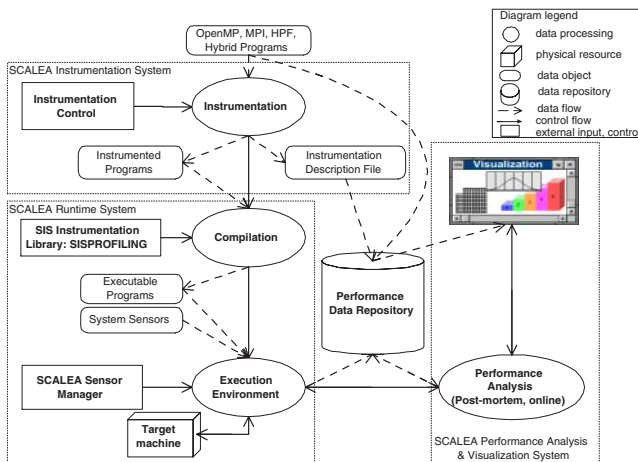


Fig. 1. Architecture of SCALEA

in the code which will collect all relevant performance information during execution of the program on a target architecture.

The SCALEA runtime system supports profiling and tracing for parallel and distributed applications, and sensors and sensor managers for capturing and managing performance data of individual computing nodes of parallel and distributed machines. The SIS profiling and tracing library collects timing, event, and counter information, as well as hardware parameters. Hardware parameters are determined through an interface with the PAPI library [2]. Various interfaces to other libraries such as TAU [6] are also supported.

The SCALEA Performance Analysis and Visualization module analyzes the raw performance data – collected during program executed and stored in the performance data repository – computes all user-requested performance metrics, and visualizes them together with the input program. Besides single-experiment analysis, SCALEA also supports multi-experiment performance analysis. The visualization engine provides a rich set of displays for performance metrics in isolation or together with the source code.

SCALEA performance data repository holds relevant information about the experiments conducted which includes raw performance data and metrics, source code, machine information, etc.

4 Experiments

In order to evaluate the usefulness of SCALEA, we present two different experiments covering a laser physics and a material science application that have been conducted on a SMP cluster architecture with 16 SMP nodes (connected by Fast-Ethernet 100Mbps and Myrinet) each of which comprises 4 Intel Pentium III 700 MHz CPUs.

4.1 3D Particle-In-Cell (3DPIC)

The 3D Particle-In-Cell application [5] simulates the interaction of high intensity ultrashort laser pulses with plasma in three dimensional geometry. This application (3DPIC) has been implemented as a Fortran90/MPI code.

We conducted several experiments by varying the machine size and by selecting the MPICH communication library for Fast-Ethernet 100Mbps. The problem size (3D geometry) has been fixed with 30 cells in x-direction (*nnx_glob*=30), 30 cells in y-direction (*nnx_glob*=30), and 100 cells in z-direction (*nnz_glob*=100). The simulation has been done for 800 time steps (*itmax*=800).

Single Experiment Analysis Mode: SCALEA provides several analyses (e.g. *Load Imbalance Analysis*, *Inclusive/Exclusive Analysis*, *Metric Ratio Analysis*, *Overhead Analysis*, *Summary Analysis*) and diagrams to support performance evaluation based on a single execution of a program. In the following we just highlight some interesting results for a single experiment of the 3DPIC code with 3 SMP nodes and 4 CPUs per node.

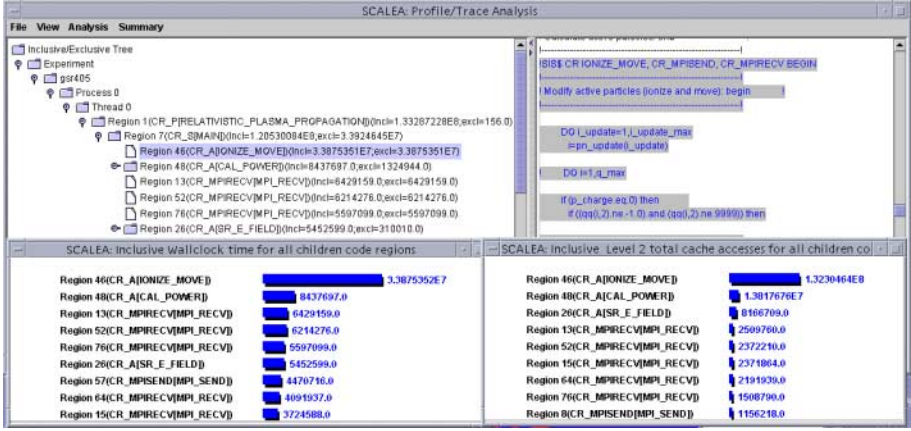


Fig. 2. Inclusive/Exclusive analysis for sub-regions of the *MAIN* program

The **Inclusive/Exclusive Analysis** is used to determine the execution time or overhead intensive code regions. The two lower-windows in Fig. 2 present the inclusive wallclock times and the number of L2 cache accesses for sub-regions of the subroutine *MAIN* executed by thread 0 in process 0 of SMP node gsr405. The most time consuming region is *IONIZE_MOVE* because it is the most computation intensive region in 3DPIC which modifies the position of the particles by solving the equation of motion $\frac{d}{dt}(m\mathbf{v}) = q(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B})$ with a forth order Runge-Kutta routine. The related source code of region *IONIZE_MOVE* is shown in the upper-right window.

The **Metric Ratio Analysis** of SCALEA is then used to examine various important metric ratios (e.g. cache miss ratio, system time/wall clock time, floating point instructions per second, etc.) of code region instance(s) in one experiment. Figure 3 shows the most critical system time/wall clock time and L2 cache misses/L2 cache accesses ratios together with the corresponding code re-

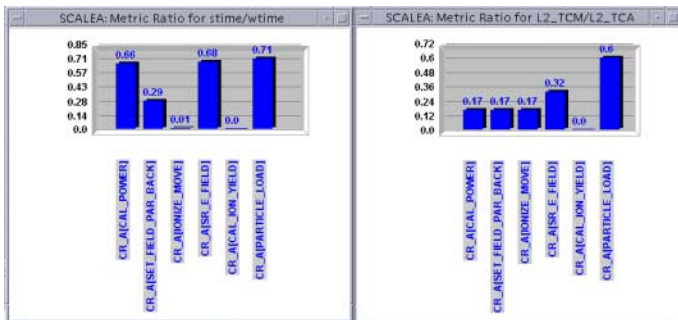


Fig. 3. Metric ratios for important code regions

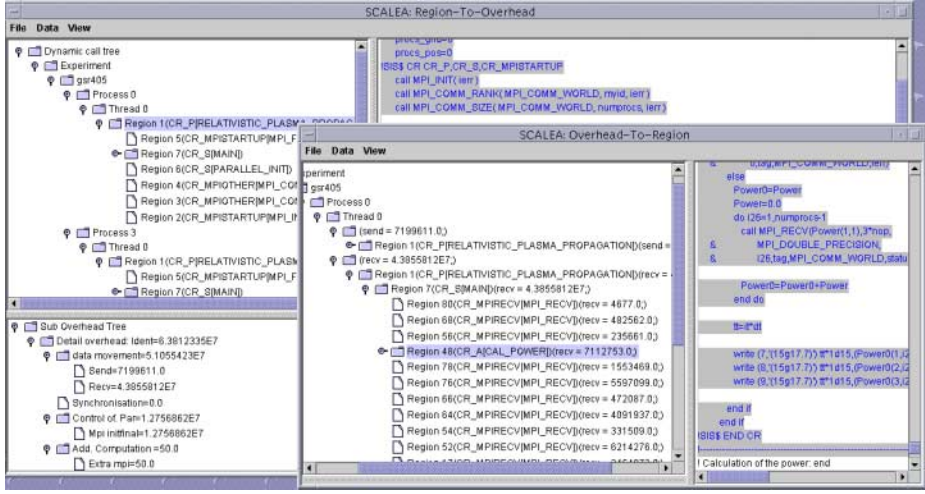


Fig. 4. Overhead-To-Region and Region-To-Overhead analysis

gions. The code regions *CAL_POWER*, *SET_FIELD_PAR_BACK*, *SR_E_FIELD*, and *PARTICLE_LOAD* imply a high system time/wall clock time ratio due to expensive MPI constructs (included in system time). Both ratios are rather low for region *IONIZE_MOVE* because this region represents the computational part without any communication (mostly user time). The code region *PARTICLE_LOAD* shows a very high L2 cache misses/L2 cache accesses ratio because it initializes all particles in the 3D volume without accessing it again (little cache reuse).

The **Overhead Analysis** is used to investigate performance overheads for an experiment based on a overhead classification [11, 10]. In Fig. 4, SCALE's *Region-To-Overhead* analysis (see the left-most window) examines the overheads of the code region instance with number 1 (thread 0, process 0, node gsr405) which corresponds to the main program. The main program is dominated by the data movement overhead (see the lower-left window) which can be further refined to the overhead associated with send and receive operations. We then use the *Overhead-To-Region analysis* to inspect regions that cause receive overhead for thread 0, process 0, and node gsr405 (see the Overhead-To-Region window in Fig. 4). The largest portion of the receive overhead in subroutine MAIN is found in region *CAL_POWER* (7.11 seconds out of 43.85 seconds execution time).

An **Execution Summary Analysis** has been employed to examine the impact of communication on the execution time of the entire program. Figure 5 and 6 depict the *execution time summary* for the experiment executed with 3 SMP nodes and with 1 SMP node (4 processors per node), respectively. In the top window of Fig. 5 each pie represents one SMP node and each pie slice value corresponds to the average value across all processes of an SMP node (min/max/average values can be selected). By clicking onto an SMP pie,

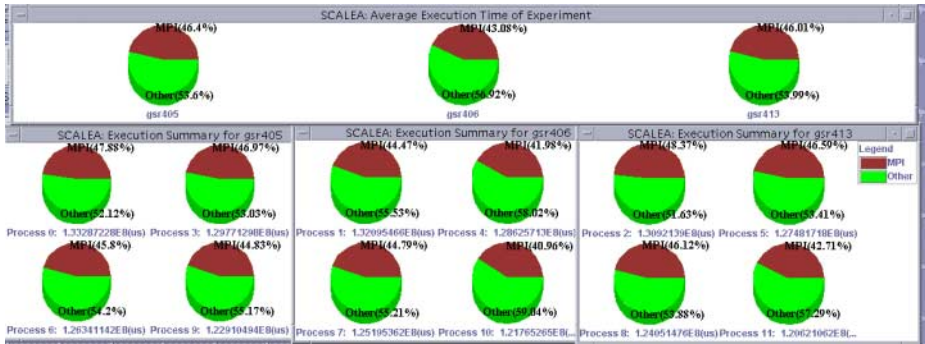


Fig. 5. Execution time summary for an experiment with 3 SMP-nodes



Fig. 6. Execution time summary for an experiment with 1 SMP-node

SCALEA displays a detailed summary for all processes in this node in the three lower windows of Fig. 5. Each pie is broken down into time spent in MPI routines and the remainder. Clearly, SCALEA indicates the dramatic increase of communication time when increasing number of SMP nodes. The MPI portion of the overall execution time corresponds approximately to 8.5% for the single SMP node version which raises to approximately 46% for 3 SMP-nodes. The experiments are based on a smaller problem size. Therefore, the communication to execution time ratio is rather high.

Multiple Experiment Analysis Mode: SCALEA provides various options to support multiple experiment performance analyses which can be applied to single or multiple region(s) ranging for single statements to the entire program.

Figure 7 visualizes the execution time and speedup/improvement of the entire 3DPIC application, respectively. With increasing machine sizes, also the system time raises (close to the user time for 25 CPUs) due to extensive communication.

Overall, 3DPIC doesn't scale well for the problem size considered because of the poor communication behavior (see Fig. 8) and also due to control of parallelism (for instance, initialization and finalization MPI operations).

4.2 LAPW0

LAPW0 [1] is a material science program that calculates the effective potential of the Kohn-Sham eigen-value problem. LAPW0 has been implemented as a Fortran90 MPI code.

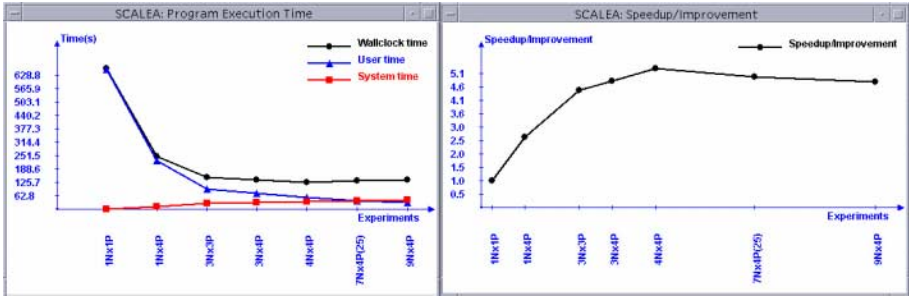


Fig. 7. Overall execution time and speedup/improvement for 3DPIC. $1N \times 4P$ means 1 SMP node with 4 processors (in case of $7N \times 4P$ only 25 processors are used)

| Experiments | 1Nx1P | 1Nx4P | 3Nx3P | 3Nx4P | 4Nx4P | 7Nx4P(25) | 9Nx4P |
|---------------------------|---------|---------|---------|---------|--------|-----------|---------|
| Data movement | 0 | 15.834 | 49.928 | 51.055 | 55.087 | 66.557 | 68.914 |
| Synchronization | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Control of parallelism | 0.027 | 3.506 | 9.257 | 12.757 | 17.396 | 28.025 | 41.12 |
| Loss of Parallelism | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Additional Overhead | 0 | 0 | 0 | 0 | 0.001 | 0 | 0 |
| Total identified overhead | 0.027 | 19.340 | 59.185 | 63.812 | 72.483 | 94.581 | 110.034 |
| Total execution time(s) | 628.676 | 237.888 | 143.476 | 133.267 | 121.73 | 129.031 | 134.267 |

Fig. 8. Performance overheads of the 3DPIC application. Note that total and unidentified overhead are missing as no sequential code version is available for the 3DPIC

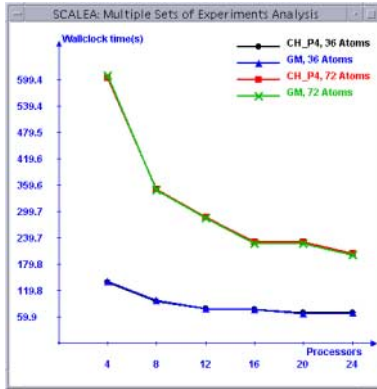


Fig. 9. Execution time of LAPW0 with 36 and 72 atoms. CH_P4 , GM means that MPICH has been used for CH_P4 (for Fast-Ethernet 100Mbps) and Myrinet, respectively

In [11] we described some results based on a single problem size and a fixed communication library and target machine network. In this section, we outline some additional performance analyses for different machine and problem sizes (36 and 72 atoms) with two different networks. The overall execution times provided by SCAEA's **Multi-Set Experiment Analysis** are shown in Fig. 10. The first observation is that changing the communication library and communication network from MPICH CH_P4 to MPICH GM does not lead to a performance improvement of the parallel LAPW0 version. The second observation is that we cannot achieve better performance by varying the number of processors from 12 to 16 and 20 to 24 processors for the experiments with 32 atoms, or by varying the processor number from 20 to 24 processors for the 72 atom experiment. In

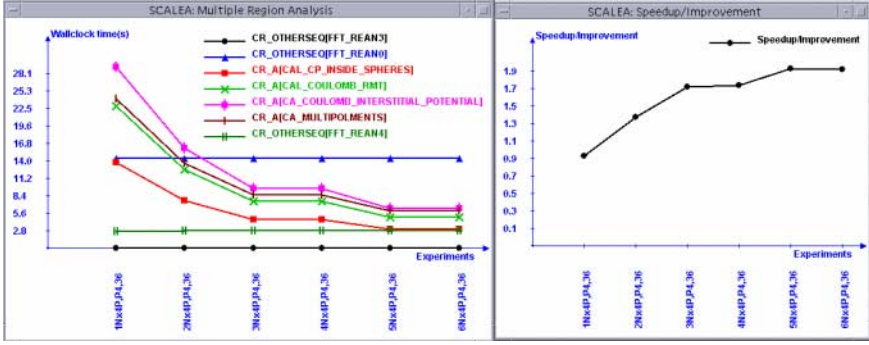


Fig. 10. Execution time of computationally intensive code regions. $1N \times 4P, P_4, 36$ means 1 SMP node with 4 processors using MPICH CH_P4 and the problem size is 36 atoms

| Experiments | 1N4P,P4,36 | 2N4P,P4,36 | 3N4P,P4,36 | 4N4P,P4,36 | 5N4P,P4,36 | 6N4P,P4,36 |
|-----------------------------|------------|------------|------------|------------|------------|------------|
| Data movement | 0.904 | 0.933 | 2.562 | 2.426 | 1.809 | 2.749 |
| Synchronization | 0 | 0 | 0 | 0 | 0 | 0 |
| Control of parallelism | 2.995 | 3.939 | 4.743 | 5.270 | 5.914 | 6.519 |
| Loss of Parallelism | 12.544 | 14.682 | 15.358 | 15.722 | 15.921 | 16.065 |
| Additional Overhead | 0 | 0 | 0 | 0 | 0 | 0 |
| Total identified overhead | 16.443 | 19.555 | 22.662 | 23.418 | 23.644 | 25.333 |
| Total unidentified overhead | 14.959 | 23.078 | 19.382 | 26.75 | 24.891 | 26.911 |
| Total overhead | 31.402 | 42.633 | 42.045 | 50.168 | 48.534 | 52.245 |
| Total execution time(s) | 137.704 | 95.784 | 77.479 | 76.744 | 69.795 | 69.962 |

Fig. 11. Performance overheads for LAPW0

order to explain this behavior we concentrate on the experiment with 36 atoms. Figure 10 visualizes the execution times for the most computational intensive code regions of LAPW0 supported by **Multiple Region Analysis**. The execution times of these code regions remain almost constant although the number of processors is increased from 12 to 24 and from 20 to 24. The LAPW0 code exposes a load balancing problem for 16, 20, and 24 processors. Moreover, code regions FFT_REAN0, FFT_REAN3, and FFT_REAN4 are executed sequentially as shown in Fig. 10. These code regions cause a large of loss of parallelism overhead (see Fig. 11).

In summary, LAPW0 scales poorly due to load imbalances and a large loss of parallelism overhead caused by the lack of parallelizing the FFT_REAN0, FFT_REAN3, and FFT_REAN4 code regions. In order to improve the performance of this application, these code regions must be parallelized as well.

5 Conclusions and Future Work

In this paper, we briefly gave an overview of SCALEA, and then described two different experiments with a laser physics and a material science code to examine the usefulness of our performance analysis approach.

Based on the flexible instrumentation mechanism and the code region performance analysis, SCALEA could determine the performance overheads that imply performance problems and relate them back to the code regions that cause them. SCALEA stores the collected performance data for all experiments in a data repository based on which a single- or multi-experiment analysis is conducted. This feature enables SCALEA to examine the scaling behavior of parallel and distributed programs. A rich set of performance diagrams is supported to filter performance data and to restrict performance analysis to relevant metrics and code regions.

SCALEA is part of the ASKALON performance tool set for cluster and Grid architectures [4] which comprises various other tools including an automatic bottleneck analysis, performance experiment and parameter study, and performance prediction tool. We are currently extending SCALEA for online monitoring of Grid applications and infrastructures.

References

- [1] P. Blaha, K. Schwarz, and J. Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, ISBN 3-9501031-0-4, 1999. 427
- [2] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceeding SC'2000*, November 2000. 424
- [3] F. Wolf and B. Mohr. Automatic Performance Analysis of SMP Cluster Applications. Technical Report Tech. Rep. IB 2001-05, Research Center Juelich, 2001. 423
- [4] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H.-L. Truong. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. www.par.univie.ac.at/project/askalon, Institute for Software Science, University of Vienna. 430
- [5] M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas*. PhD thesis, Vienna University of Technology, 2001. 424
- [6] Allen Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. In *In G. Kotsis and P. Kacsuk (Eds.), Third International Austrian/Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, pages 37–46. Kluwer Academic Publishers, Sept. 2000. 422, 424
- [7] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995. 422
- [8] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996. 422
- [9] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993. 422

- [10] Hong-Linh Truong and Thomas Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International EuroPar Conference (EuroPar 2002)*, Lecture Notes in Computer Science, Paderborn, Germany, August 2002. Springer-Verlag. 422, 423, 426
- [11] Hong-Linh Truong, Thomas Fahringer, Georg Madsen, Allen D. Malony, Hans Moritsch, and Sameer Shende. On Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Proceeding of the 9th IEEE/ACM High-Performance Networking and Computing Conference (SC'2001)*, Denver, USA, November 2001. 422, 423, 426, 428
- [12] T. Cortes V. Pillet, J. Labarta and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *WoTUG-18*, pages 17–31, Manchester, April 1995. 422

A Performance Study of Load Balancing Strategies for Approximate String Matching on an MPI Heterogeneous System Environment

Panagiotis D. Michailidis and Konstantinos G. Margaritis

Parallel and Distributed Processing Laboratory
Department of Applied Informatics, University of Macedonia
156 Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece
{panosm,kmarg}@uom.gr
<http://macedonia.uom.gr/~{panosm,kmarg}>

Abstract. In this paper, we present three parallel approximate string matching methods on a parallel architecture with heterogeneous workstations to gain supercomputer power at low cost. The first method is the static master-worker with uniform distribution strategy, the second one is the dynamic master-worker with allocation of subtexts and the third one is the dynamic master-worker with allocation of text pointers. Further, we propose a hybrid parallel method that combines the advantages of static and dynamic parallel methods in order to reduce the load imbalance and communication overhead. This hybrid method is based on the following optimal distribution strategy: the text collection is distributed proportional to workstation's speed. We evaluated the performance of four methods with clusters 1, 2, 4, 6 and 8 heterogeneous workstations. The experimental results demonstrate that the dynamic allocation of text pointers and hybrid methods achieve better performance than the two original ones.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to information and multimedia retrieval, computational biology, pattern recognition, Web search engines and text mining. It is defined as follows: given a large text collection $t = t_1t_2...t_n$ of length n , a short pattern $p = p_1p_2...p_m$ of length m and a maximal number of errors allowed k , we want to find all text positions where the pattern matches the text up to k errors. Errors can be substituting, deleting, or inserting a character.

In the on-line version of the problem, it is possible to preprocess the pattern but not the text collection. The classical solution involves dynamic programming and needs $O(mn)$ time [14]. Recently, a number of sequential algorithms improved the classical time consuming one; see for instance the surveys [7,11]. Some of them are sublinear in the sense that they do not inspect all the characters of the text collection.

We are particularly interested in information retrieval, where current free text collections is normally so very large that even the fastest on-line sequential algorithms are not practical, and therefore the parallel and distributed processing becomes necessary. There are two basic methods to improve the performance of approximate string matching on large text collections: one is based on the fine-grain parallelization of the approximate string matching algorithm [2,12,13,6,4,5] and the other is based on the distribution of the computation of character comparisons on supercomputers or network of workstations. As far as the second method, is concerned distributed implementations of approximate string matching algorithm are not available in the literature. However, we are aware of few attempts for implementing other similar problems on a cluster of workstations. In [3] a exact string matching implementation have been proposed and results are reported on a transputer based architecture. In [9,10] a exact string matching algorithm was parallelized and modeled on a homogeneous platform giving positive experimental results. Finally, in [5,16] presented parallelizations of a biological sequence analysis algorithm on a homogenous cluster of workstations and on an Intel iPSC/860 parallel computer respectively. However, the general efficient algorithms for the master-worker paradigm on heterogeneous clusters have been widely developed in [1].

The main contribution of this work is three low-cost parallel approximate string matching approaches that can search in very large free textbases on inexpensive cluster of heterogeneous PCs or workstations running Linux operating system. These approaches are based on master-worker model using static and dynamic allocation of the text collection. Further, we propose a hybrid parallel approach that combines the advantages of three previous parallel approaches in order to reduce the load imbalance and communication overhead. This hybrid approach is based on the following optimal distribution strategy: the text collection is distributed proportional to workstation's speed. The four approaches are implemented using the MPI library [15] over a cluster of heterogeneous workstations. To the best of our knowledge, this is the first attempt the implementation of approximate string matching application using static and dynamic load balancing strategies on a network of heterogeneous workstations.

2 MPI Master-Worker Implementations of Approximate String Matching

We follow master-worker programming model to develop our parallel and distributed approximate string matching implementations under MPI library [15].

2.1 Static Master-Worker Implementation

In order to present the static master-worker implementation we make the following assumptions: First, the workstations are numbered from 0 to $p - 1$, second, the documents of our text collection are distributed among the various workstations and stored on their local disks and finally, the pattern and the number

of errors k are stored in the main memory to all workstations. The partitioning strategy of this approach is to partition the entire text collection into a number of the subtext collections according to the number of workstations allocated. The size of each subtext collection should be equal to the size of the text collection divided by the number of allocated workstations. Therefore, the static master-worker implementation that is called P1 is composed of four phases. In first phase, the master broadcasts the pattern string and the number of errors k to all workers. In second phase, each worker reads its subtext collection from the local disk in the main memory. In third phase, each worker performs character comparisons using a local sequential approximate string matching algorithm to generate the number of occurrences. In fourth phase, the master collects the number of occurrences from each worker.

The advantage of this simple approach is low communication overhead. This advantage was achieved, a priori, by the search computation, assigning each worker to search its own subtext independently without have to communicate with the other workers or the master. However, the main disadvantage is the possible load imbalance because of the poor partitioning technique. In other words, there is a significant idle time for faster or more lightly loaded workstations on a heterogeneous environment.

2.2 Dynamic Master-Worker Implementations

In this subsection, we implement two versions of the dynamic master-worker model. The first version is based on the dynamic allocation of subtexts and the second one is based on the dynamic allocation of text pointers.

Dynamic Allocation of Subtexts The dynamic master-worker strategy that we adopted is a known parallelization strategy and is known as "workstation farm". Before, we present the dynamic implementation we make the following assumption: the entire text collection is stored on the local disk of the master workstation. The dynamic master-worker implementation that is called P2 is composed of six phases. In first phase, the master broadcasts the pattern string and the number of errors k to all workers. In second phase, the master reads from the local disk the several chunks of the text collection. The size of each chunk (sb) is an important parameter which can be affect the overall performance. More specifically, this parameter is directly related to the I/O and communication factors. We selected several sizes of each chunk in order to find the best performance as we presented in our experiments [8]. In third phase, the master sends the first chunks of the text collection to corresponding worker workstations. In fourth phase, each worker workstation performs a sequential approximate string matching algorithm between the corresponding chunk of text and the pattern in order to generate the number of occurrences. In fifth phase, each worker sends the number of occurrences back to master workstation. In sixth phase, if there are still any chunks of the text collection left, the master reads and distributes next chunks of the text collection to workers and loops back to fourth phase.

The advantage of this dynamic approach is low load imbalance, while the disadvantage is higher inter-workstation communication overhead.

Dynamic Allocation of Text Pointers Before, we present the dynamic implementation with the text pointers we make the following assumptions: First, the complete text collection is stored on the local disks of all workstations and second, the master workstation has a text pointer that shows the current position in the text collection. The dynamic allocation of text pointers that is called P3 is composed of six phases. In first phase, the master broadcasts the pattern string and the number of errors k to all workers. In second phase, the master sends the first text pointers to corresponding workers. In third phase, each worker reads from the local disk the sb characters of text starting from the pointer that receives. In fourth phase, each worker performs a sequential approximate string matching procedure between the corresponding chunk of text and the pattern in order to generate the number of occurrences. In fifth phase, each worker sends the result back to master. In sixth phase, if the text pointer does not reach the end of the text, then master updates the text pointers for the next position of next chunks of text and sends the pointers to workers and loops back to third phase.

The advantage of this simple implementation is that reduces the inter workstation communication overhead since each workstation in this scheme has an entire copy of the text collection on the local disk. However, this scheme requires more local space (or disk) requirements, but the size of the local disk in parallel and distributed architectures is large enough.

2.3 Hybrid Master-Worker Implementation

Here, we develop a hybrid master-worker implementation that combines the advantages of static and dynamic approaches in order to reduce the load imbalance and communication overhead. This implementation is based on the optimal distribution strategy of the text collection that is performed statically. In the following subsection, we describe the optimal text distribution strategy and its implementation.

Text Distribution and Load Balancing To avoid the slowest workstations to determine the parallel string matching time, the load should be distributed proportionally to the capacity of each workstation. The goal is to assign the same amount of time, which may not correspond to the same amount of the text collection.

A balanced distribution is achieved by a static load distribution made prior to the execution of the parallel operation. To achieve a good balanced distribution among heterogeneous workstations, the amount of text distributed to each workstation should be proportional to its processing capacity compared to the entire network:

$$l_i = \frac{S_i}{\sum_{j=0}^{p-1} S_j} \quad (1)$$

where S_j is the speed of the workstation j . Therefore, the amount of the text collection that is distributed to each workstation M_i ($1 \leq i \leq p$) is $l_i * n$, where n is the length of the complete text collection.

The hybrid implementation that is called P4 is same as the P1 implementation but we use the optimal distribution method instead of the uniform distribution one. The four entire parallel implementations are constructed so that alternative sequential approximate string matching algorithms can be substituted quite easily [7, 11]. In this paper, we use the classical SEL dynamic programming algorithm [14].

3 Experimental Results

In this section, we discuss the experimental results for the performance of four parallel and distributed algorithms. These algorithms are implemented in C programming language using the MPI library [15].

3.1 Experimental Environment

The target platform for our experimental study is a cluster of heterogeneous workstations connected with 100 Mb/s Fast Ethernet network. More specifically, the cluster consists of 4 Pentium MMX 166 MHz with 32 MB RAM and 6 Pentium 100 MHz with 64 MB RAM. A Pentium MMX is used as master workstation. The average speeds of the two types of workstations, Pentium MMX and Pentium, for the four implementations are listed in Table 1. The MPI implementation used on the network is MPICH version 1.2. During all experiments, the cluster of workstations was dedicated. Finally, to get reliable performance results 10 executions occurred for each experiment and the reported values are the average ones. The text collection we used was composed of documents, which were portion of the various web pages.

3.2 Experimental Results

In this subsection, we present the experimental results concluding from two sets of experiments. For the first experimental setup, we study the performance of four master-worker implementations P1, P2, P3 and P4. For the second experimental setup, we examine the scalability issue of our implementations by doubling the text collection.

Table 1. Average speeds (in chars per sec) of the two types of workstations

| Application | Pentium MMX | Pentium |
|-------------|-------------|-------------|
| P1,P4 | 3693675.74 | 2200079.448 |
| P2 | 3753988.176 | 2194043.93 |
| P3 | 3737505.55 | 2197613.22 |

Comparing the Four Types of Approximate String Matching Implementations Before we present the results for four methods, we determined from the extensive experimental study [8] that the block size nearly $sb=100,000$ characters produces optimal performance for two dynamic master-worker methods P2 and P3, later experiments are all performed using this optimal value for the P2 and P3. Further, from [8] we observed that the worst performance is obtained for very small and large values of block size. This is because small values of block size increase the inter-workstation communication, while large values of block size produce poorly balanced load.

Figures 1 and 2 show the execution times and the speedup factors with respect to the number of workstations respectively. It is important to note that the execution times and the speedups, which are plotted in Figures 1 and 2 are result of average for five pattern lengths ($m=5, 10, 20, 30$ and 60) and four values of the number of errors ($k=1, 3, 6$ and 9). The speedup of a heterogeneous computation is defined as the ratio of the sequential execution time on the fastest workstation to the parallel execution time across the heterogeneous cluster. To have a fair comparison in terms of speedup, one defines the system computing power, which considers the power available instead of the number of workstations. The system computing power defines as follows: $\sum_{i=0}^{p-1} S_i/S_0$ for p workstations used, where S_0 is the speed of the master workstation.

As we have expected, performance results show that the P1 implementation using static load balancing strategy is less effective than the other three implementations in case of heterogeneous network. This fact due to the presence of waiting time associated to communications. In other words, the slowest workstation is always the latest one in string matching computation. Further, the P2 implementation using dynamic allocation of subtexts produces better results than the P1 one in case of heterogeneous cluster. Finally, the experimental results show that the P3 and P4 implementations seem to have the best performance compared with the others in case of heterogeneous cluster. These implementations give smaller execution times and higher speedups than in case of using

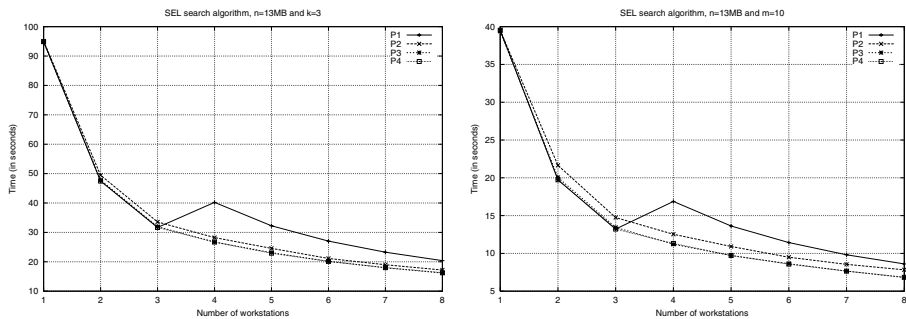


Fig. 1. Experimental execution times (in seconds) for text size of 13MB and $k=3$ using several pattern lengths (left) and $m=10$ using several values of k (right)

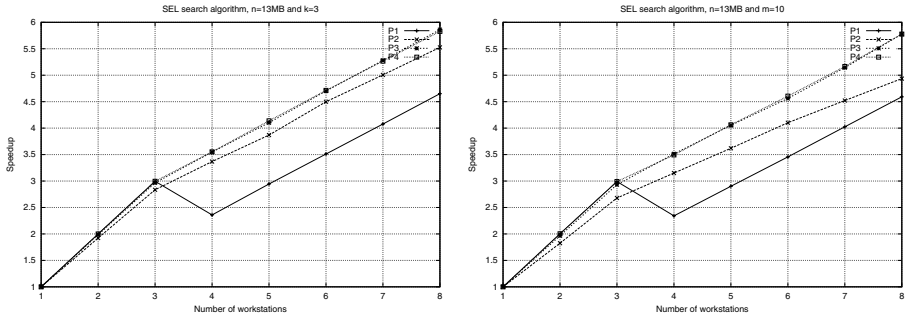


Fig. 2. Speedup of parallel approximate string matching with respect to the number of workstations for text size of 13MB and $k=3$ using several pattern lengths (left) and $m=10$ using several values of k (right)

the P1 and P2 ones when the network becomes heterogeneous, i.e. after the 3rd workstation.

We now examine the performance of the P2, P3 and P4 parallel implementations. From the results, we see a clear reduction in the computation time of the algorithm when we use the three parallel implementations. For instance, with $k=3$ and several pattern lengths, we reduce the average computation time from 95.085 seconds in the sequential version to 17.206, 16.176 and 16.040 seconds in the distributed implementations P2, P3 and P4 respectively using 8 workstations. In other words, from the Figure 1 we observe that for constant total text size there is an expected inverse relation between the parallel execution times and the number of workstations. Further, the three master-worker implementations achieve reasonable speedups for all workstations. For example, with $k=3$ and several pattern lengths, we had an increasing speedup curves up to about 5.52, 5.86 and 5.91 in distributed methods P2, P3 and P4 respectively on the 8 workstations which had the computing power of 5.92, 5.93 and 5.97.

Scalability Issue To study the scalability of three proposed parallel implementations P2, P3 and P4, we setup the experiments in the following way. We simply double the old text size two times. This new text collection is around 27MB. Results from these experiments have been depicted in Figures 3 and 4. The results show that the three parallel implementations still scale well though the problem size has been increased two times (i.e. doubling the text collection). The average execution times for $k=3$ and several pattern lengths similarly decrease to 34.063, 32.298 and 32.150 seconds for the P2, P3 and P4 implementations respectively when the number of workstations have been added to 8. Moreover, speedup factors of three methods also linearly increase when the workstations are increased. Finally, the best performance results are obtained with the P3 and P4 load balancing methods.

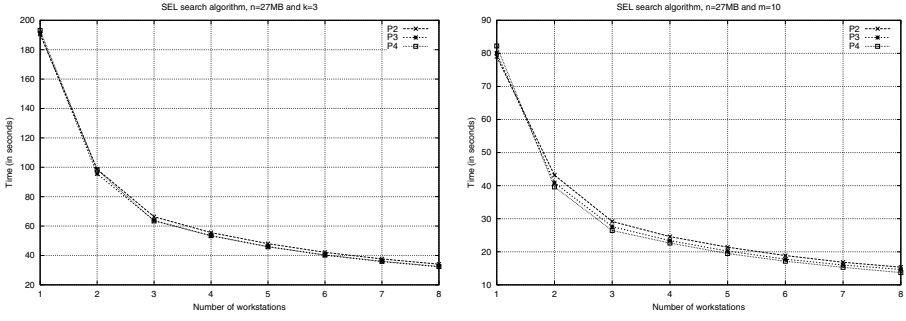


Fig. 3. Experimental execution times (in seconds) for text size of 27MB and $k=3$ using several pattern lengths (left) and $m=10$ using several values of k (right)

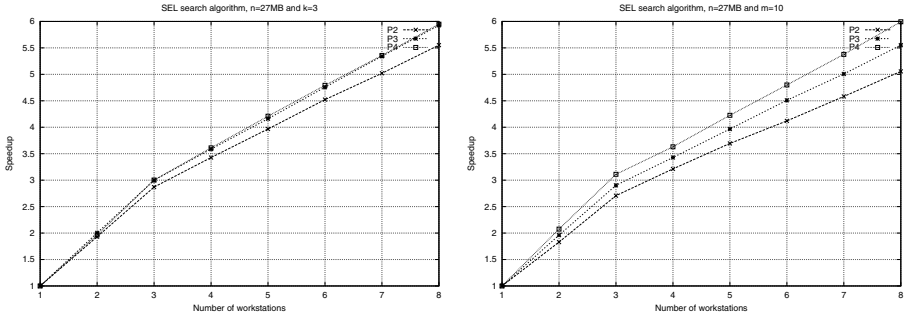


Fig. 4. Speedup of parallel approximate string matching with respect to the number of workstations for text size of 27MB and $k=3$ using several pattern lengths (left) and $m=10$ using several values of k (right)

4 Conclusions

In this paper, we have presented four parallel and distributed approximate string matching implementations and the performance results on a low-cost cluster of heterogeneous workstations. We have observed from this extensive study that the P3 and P4 implementations produce better performance results in terms execution times and speedups than the others. Higher gains in performance are expected for a larger number of varying speed workstations in the network.

Variants of the approximate string matching algorithm can directly be implemented on a cluster of heterogeneous workstations using the four text distribution methods reported here. We plan to develop a theoretical performance model in order to confirm the experimental behaviour of four implementations on a heterogeneous cluster. Further, this model can be used to predict the execution time and similar performance metrics for the four approximate string matching implementations on larger clusters and problem sizes.

References

1. O. Beaumont, A. Legrand and Y. Robert, The master-slave paradigm with heterogeneous processors, Report LIP RR-2001-13, 2001. 433
2. H.D. Cheng and K.S. Fu, VLSI architectures for string matching and pattern matching, *Pattern Recognition*, vol. 20, no. 1, pp. 125-141, 1987. 433
3. J. Cringean, R. England, G. Manson and P. Willett, Network design for the implementation of text searching using a multicomputer, *Information Processing and Management*, vol. 27, no. 4, pp. 265-283, 1991. 433
4. D. Lavenier, Speeding up genome computations with a systolic accelerator, *SIAM News*, vol. 31, no. 8, pp. 6-7, 1998. 433
5. D. Lavenier and J.L. Pacherie, Parallel processing for scanning genomic databases, in *Proc. PARCO'97*, pp. 81-88, 1997. 433
6. K.G. Margaritis and D.J. Evans, A VLSI processor array for flexible string matching, *Parallel Algorithms and Applications*, vol. 11, no. 1-2, pp. 45-60, 1997. 433
7. P.D. Michailidis and K.G. Margaritis, On-line approximate string searching algorithms: Survey and experimental results, *International Journal of Computer Mathematics*, vol. 79, no. 8, pp. 867-888, 2002. 432, 436
8. P.D. Michailidis and K.G. Margaritis, Performance evaluation of load balancing strategies for approximate string matching on a cluster of heterogeneous workstations, Tech. Report, Dept. of Applied Informatics, University of Macedonia, 2002. 434, 437
9. P.D. Michailidis and K.G. Margaritis, String matching problem on a cluster of personal computers: Experimental results, in *Proc. of the 15th International Conference Systems for Automation of Engineering and Research*, pp. 71-75, 2001. 433
10. P.D. Michailidis and K.G. Margaritis, String matching problem on a cluster of personal computers: Performance modeling, in *Proc. of the 15th International Conference Systems for Automation of Engineering and Research*, pp. 76-81, 2001. 433
11. G. Navarro, A guided tour to approximate string matching, *ACM Computer Surveys*, vol. 33, no. 1, pp. 31-88, 2001. 432, 436
12. N. Ranganathan and R. Sastry, VLSI architectures for pattern matching, *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 8, no. 4, pp. 815-843, 1994. 433
13. R. Sastry, N. Ranganathan and K. Remedios, CASM: a VLSI chip for approximate string matching, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 824-830, 1995. 433
14. P.H. Sellers, The theory and computations of evolutionary distances: pattern recognition, *Journal of Algorithms*, vol. 1, pp. 359-373, 1980. 432, 436
15. M. Snir, S. Otto, S. Huss-Lederman, D.W. Walker and J. Dongarra, MPI: The complete reference, The MIT Press, Cambridge, Massachusetts, 1996. 433, 436
16. T.K. Yap, O. Frieder and R.L. Martino, Parallel computation in biological sequence analysis, *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 283-293, 1998. 433

An Analytical Model for Pipeline Algorithms on Heterogeneous Clusters^{*}

F. Almeida, D. González, L.M. Moreno, and C. Rodríguez

Dpto. Estadística, I.O. y Computación, La Laguna University, Spain
{falmeida,dgonmor,lmmoreno,casiano}@ull.es

Abstract. The performance of a large class of virtual pipeline algorithms on heterogeneous networks is studied. The word heterogeneity refers here both to the processing and communication capabilities. To balance the differences among processors requires a vectorial combination of block and cyclic mappings, assigning different number of virtual processes per processor. Following a progressive approach, we derive a formula that predicts the performance for the general case. The computational results carried out on a heterogeneous cluster of PCs prove the effectiveness of the approach and the accuracy of the predictions.

1 Introduction

The availability of modern networking technologies and cost-effective commodity computing boxes has attracted the attention to high performance developments towards Networks Of Workstations (NOW). The NOW systems comprise clusters of PCs/workstations connected over the Local Area Networks (LAN) and provide an attractive price to performance ratio. Many research projects have been carried out to provide accurate prediction models for NOWs. However, most of these projects focus on homogeneous NOWs, systems comprising of similar kinds of PCs/workstations connected over a single network architecture. The inherent scalability of the NOW environment combined with the commodity nature of the PCs/workstations and networking equipment is forcing the NOW systems to become heterogeneous in nature (HNOW).

Some authors [3] consider heterogeneity due to the difference in processing speeds of workstations. In [2] it has been proved that the heterogeneity in the speed of workstations can have a significant impact on the communication send/recv overhead. Following this idea, we consider also the heterogeneity in the network due to the difference in communication speeds of the workstations. The development of models providing accurate predictions, while efficiently exploiting the heterogeneity on HNOWs, appears now as a new challenge. Considering heterogeneity due both to computation and communication, provides a very general overview of the system and the resulting models have a wide range of applicability.

^{*} Authors are listed in alphabetical order. This work has been partially supported by: Spanish CICYT under grant TIC-1999-0754-03 (MALLBA).

The pipeline paradigm so extensively studied in homogeneous architectures [1, 4], has not attracted the same attention of researchers towards heterogeneous platforms. The prediction on a heterogeneous platform must take into account that the computation on a pipeline is coerced by the slowest slot and this fact should be modeled. We present an analytical model that allows the prediction of block-cyclic assignment policies of pipelines on heterogeneous systems. The model considers heterogeneity due both to computation and to communication. The computational results prove the efficiency of the scheme and the accuracy of the predictions. The predictive ability of the proposed model constitutes the basis for further studies on the pipeline paradigm.

The paper is organized as follows. Section 2 describes the testbed used to verify our proposed model and the measured values of point-to-point communications. Section 3 goes into the family of pipeline problems considered, pointing out the drawbacks in the heterogeneous contexts. We have performed the modelization of the pipeline paradigm in three steps: first in section 4 is considered the case with an infinite number of processors available and in section 5 the pure cyclic mapping and the general block cyclic mapping are described. The correctness of the model that we develop in this sections is validated by comparing the results predicted by the model with the results gathered from our experimental testbed for different system configurations and problem sizes. The paper is finished in section 6 with some concluding remarks and future lines of work.

2 Measurement of Communication Parameters

A cluster with two groups of processors has been used, the first one composed by four ADM Duron (tm) 800 MHz PCs with 256 MB memory. These machines are referred as fast nodes. The second group of processors, the slow nodes, comprises four AMD-K6 (tm) 501 MHz PCs with 256 MB memory. All machines run Debian Linux Operative System. They are connected through a 100 Mbit/sec. fast Ethernet switch.

We characterize point to point communications on the various type of nodes in the heterogeneous network. For the sake of the simplicity we assume that the cost of a communication can be modeled by the classical linear model $\beta + \tau w$, β and τ stands for the latency and per-word transfer time respectively and w represents the number of bytes to be sent. The parameters β and τ are measured using a ping-pong experiment. The blocking MPI_Send and MPI_Receive calls

Table 1. Latency and Per-Word Transfer Time in our testbed

| Sender Type | Receiver Type | β | τ |
|-------------|---------------|-------------|-------------|
| Fast | Fast | 0.000161395 | 7.3125E-07 |
| Fast | Slow | 0.000173219 | 7.64322E-07 |
| Slow | Slow | 0.000129819 | 7.96033E-07 |

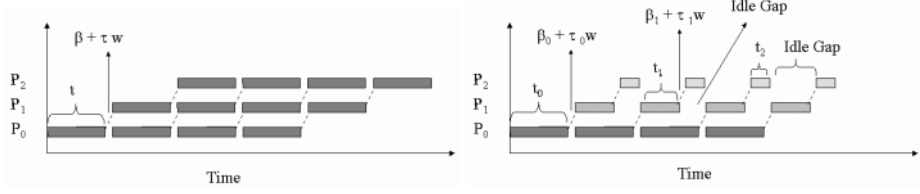


Fig. 1. Timing Diagrams: Left - Homogeneous pipeline. Right - Heterogeneous pipeline

are used. The linear fit of the sending overhead is plotted as a function of the message sizes for combinations of both slow and fast nodes (table 1). Since the parameters are measured using a ping-pong the values obtained for the Fast-Slow, Slow-Fast combinations are equal.

3 Problem Statement

We abstract a pipeline with n stages assuming that every virtual processor i of the pipe executes a M iteration loop similar to the following:

```

j = 0;
while (j < M) {
    receive ();
    compute ();
    send (item_w);
    j++;
}
    
```

Figure 1 represents the typical timing diagram for a homogeneous pipeline. The filled rectangles represent the time t that a processor spends computing on one iteration and the dotted lines $(\beta + \tau w)$ represent the communication time between computations. This regular pattern that appears in the homogeneous case has been extensively studied ([1, 4]). The heterogeneous case is far from being so considered in the literature. Figure 1 also shows the timing diagram for a pipeline with $p = 3$ processors each one with different computational power. P_0 is the slowest one and P_2 is the faster one. The filled rectangles represent the time t_i that processor i spends computing on one iteration and the dotted lines $(\beta_i + \tau_i w)$ represent the communication time between computations for processors $i, i + 1$. The illustration shows how the computation is coerced by the slowest processor.

To validate our model the computation between communications has been considered to be a synthetic constant loop. In our testbed, the values measured for the Slow and Fast processors are $t_S = 0.136$ and $t_F = 0.0726$ respectively.

4 The Model: Without Virtualization

Let us consider first that the number of physical processors p is equal to the number of stages of the pipeline n , $p = n$. Obviously, processor i of the pipeline starts the computation at time $\sum_{j=0}^{i-1} (t_j + \beta_j + \tau_j w)$.

Lets denote by T_s the start up time for the processor $p - 1$, i. e., $T_s = \sum_{i=0}^{p-2} (t_i + \beta_i + \tau_i w)$, and let i_0 denote the slowest processor in the pipe, i.e., the first processor in the pipe such that $(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) \geq \max_{i=0}^{p-1} (t_i + \beta_i + \tau_i w)$.

Proposition 1 *The total running time of the pipeline is:*

$$T = \begin{cases} T_s + Mt_{p-1} & \text{if } t_{p-1} \geq \max_{i=0}^{p-2} (t_i + \beta_i + \tau_i w) \\ \sum_{i=0}^{i_0-1} (t_i + \beta_i + \tau_i w) + M(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + \sum_{i=i_0+1}^{p-2} (t_i + \beta_i + \tau_i w) + t_{p-1} & \text{otherwise.} \end{cases}$$

Proof: It can be easily deduced from fig. 2

Proposition 2 *After being started, processor $p - 1$ finishes its computation at time*

$$T_c = \begin{cases} Mt_{p-1} & \text{if } t_{p-1} \geq \max_{i=0}^{p-2} (t_i + \beta_i + \tau_i w) \\ Mt_{p-1} + (M - 1)(t_{i_0} - t_{p-1} + \beta_{i_0} + \tau_{i_0} w) & \text{otherwise.} \end{cases}$$

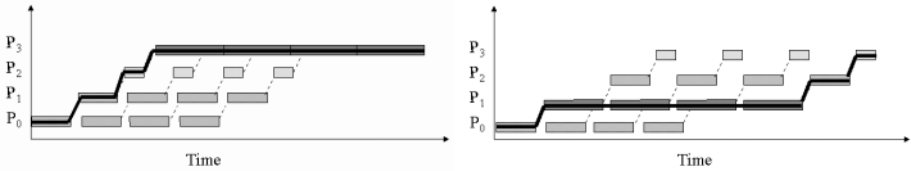


Fig. 2. Timing Diagrams. The bolded line depicts the critical path. Left - $t_{p-1} \geq \max_{i=0}^{p-2} (t_i + \beta_i + \tau_i w)$. Right - The slowest processor is in the middle of the pipeline

Proof: The case $t_{p-1} \geq \max_{i=0}^{p-2} (t_i + \beta_i + \tau_i w)$ is trivial.

Lets prove it for the other case: By proposition 1, the total runing time of the pipeline is

$$\sum_{i=0}^{i_0-1} (t_i + \beta_i + \tau_i w) + M(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + \sum_{i=i_0+1}^{p-2} (t_i + \beta_i + \tau_i w) + t_{p-1} =$$

$$\sum_{i=0}^{p-2} (t_i + \beta_i + \tau_i w) + (M-1)(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + t_{p-1} =$$

$$T_s + (M-1)(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + t_{p-1}$$

Since T_s is the time to start up the last processor, the running time for processor $p-1$ is $(M-1)(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + t_{p-1} = Mt_{p-1} + (M-1)(t_{i_0} - t_{p-1} + \beta_{i_0} + \tau_{i_0} w)$ and the theorem is proved.

As a trivial consequence of proposition 2 we have:

Theorem 1 *The total running time for a pipeline with $p = n$ processor is: $T = T_s + T_c$.*

Table 2 presents the actual times (columns labeled *Real Time*), the model times (columns labeled *Model*, computed according to theorem 1) and the relative

Table 2. Accuracy of the Model

| | P = 4 - N = 4 - M = 500 | | | P = 4 - N = 40 - M = 500 | | |
|-----------------|-------------------------|--------|-----------|--------------------------|---------|-----------|
| Configuration | Real Time | Model | Error | Real Time | Model | Error |
| F S F S | 69.535 | 68.439 | -1.58E-02 | 685.834 | 681.994 | -5.60E-03 |
| S F S F | 68.815 | 68.375 | -6.39E-03 | 712.818 | 681.930 | -4.33E-02 |
| F F S S | 69.064 | 68.439 | -9.05E-03 | 685.867 | 681.993 | -5.65E-03 |
| S S F F | 69.111 | 68.375 | -1.06E-02 | 706.692 | 681.930 | -3.50E-02 |
| | P = 6 - N = 6 - M = 500 | | | P = 6 - N = 60 - M = 500 | | |
| Configuration | Real Time | Model | Error | Real Time | Model | Error |
| F S F S F S | 70.093 | 68.648 | -2.06E-02 | 686.272 | 682.203 | -5.93E-03 |
| S F S F S F | 69.509 | 68.584 | -1.33E-02 | 713.228 | 682.139 | -4.36E-02 |
| F F F S S S | 69.490 | 68.648 | -1.21E-02 | 686.165 | 682.202 | -5.78E-03 |
| S S S F F F | 69.689 | 68.584 | -1.59E-02 | 707.203 | 682.139 | -3.54E-02 |
| | P = 8 - N = 8 - M = 500 | | | P = 8 - N = 80 - M = 500 | | |
| Configuration | Real Time | Model | Error | Real Time | Model | Error |
| F S F S F S F S | 70.695 | 68.857 | -2.60E-02 | 687.158 | 682.412 | -6.91E-03 |
| S F S F S F S F | 70.008 | 68.794 | -1.73E-02 | 714.083 | 682.348 | -4.44E-02 |
| F F F F S S S S | 69.982 | 68.857 | -1.61E-02 | 686.770 | 682.411 | -6.35E-03 |
| S S S S F F F F | 70.154 | 68.793 | -1.94E-02 | 708.064 | 682.348 | -3.63E-02 |

errors for different configurations of slow (S) and faster (F) machines. The most remarkable fact is the low error percentages (below 2.6%). The error presents a constant behavior on the number of processors. A second noticeable fact is the relative independence of model and actual times on the slow-fast processor combination. This behavior agrees with theorem 1, since the processor permutation only has influence in the value of T_s .

5 The Model: Pure Cyclic Mapping and Block Cyclic Mapping

Let us consider now that the number of stages of the pipeline n is greater than the number p of processors available, i.e., $p < n$. Therefore, it is necessary to assign the stages of the pipeline between the available processors. The classical technique consists of partitioning the set of stages following a mixed block-cyclic mapping depending on the grain G of stages assigned to each processor. In the heterogeneous case a different granularity G_i should be considered for each different processor. It is also 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. The easiest case where $G = 1$ and $B = 1$ corresponding to a pure cyclic mapping without buffering will be first approached, and later the general case will be developed. In the cyclic case, the set of stages is partitioned into $NumBands = \frac{n}{p}$ number of bands sets with the same cardinal as the number p of processors, in such a way that stage $q \in Band(i)$ if and only if $\frac{q}{p} = i$. Stage q is processed by processor $q \bmod p$.

Let us denote by $T'_s = \sum_{i=0}^{p-1} (t_i + \beta_i + \tau_i w) = T_s + t_{p-1} + \beta_{p-1} + \tau_{p-1} w$ the

start up time between bands and by $T'_c = M(t_{p-1} + \beta_{p-1} + \tau_{p-1} w + \max_{i=0}^{p-1} (t_i + \beta_i + \tau_i w - (t_{p-1} + \beta_{p-1} + \tau_{p-1} w)))$ the computation time for the processor $p - 1$ in an intermediate band. T'_c includes the time to compute and send the items plus all the idle gaps. The following theorem characterizes the timing function on a pure cyclic mapping without buffering.

Theorem 2 *The total running time on a pure cyclic mapping with n stages and p processors where $p < n$ is determined by:*

$$T = \begin{cases} T_1 = T_s + (NumBands - 1)T'_c + T_c & \text{if } T'_c > T'_s \\ T_2 = (NumBands - 1)T'_s + T_s + T_c & \text{if } T'_s > T'_c \end{cases}$$

Proof: By induction on the number of bands:

If $NumBands = 2$ the proof can be easily deduced from figure 3.

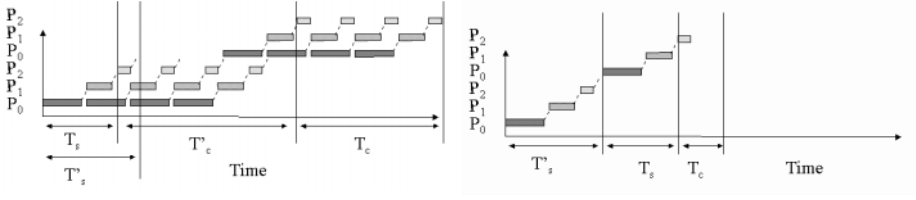


Fig. 3. Timing Diagrams: Left - $T'_c > T'_s$. Right - $T'_c < T'_s$

Let us assume that the theorem is true for $Bands = (NumBands - 1)$ and prove it for $Bands = NumBands$

$$\begin{aligned} \text{If } T'_c > T'_s &\Rightarrow T_{NumBands} = T_{NumBands-1} + T'_c = \\ T_s + (NumBands - 2)T'_c + T_c + T'_c &= T_s + (NumBands - 1)T'_c + T_c \end{aligned}$$

$$\begin{aligned} \text{If } T'_s > T'_c &\Rightarrow T_{NumBands} = T_{NumBands-1} + T'_s = \\ (NumBands - 2)T'_s + T_s + T_c + T'_s &= (NumBands - 1)T'_s + T_s + T_c \end{aligned}$$

Table 2 presents also the computational results for the pure cyclic mapping. The introduction of the round robin mapping does not introduce any increase in the errors.

Let us consider now the more general case where a block-cyclic mapping will be performed. An assignment of G_i virtual processes to processor i will be assumed and the communication buffer between processors will accept packets of size B . In this context, processor i will compute $G_i B$ items on each iteration and every communication involves B items of size w . According to the notation used along the paper, processor i will spend in each iteration $G_i B t_i$ computation time and $\beta_i + \tau_i B w$ communication time, in a pipeline with $n' = \frac{pn}{p-1}$ stages.

By analogy with the cyclic case we have the following definitions:

The slowest processor i_0 will be characterized by the first processor in the pipe such that $(G_{i_0} B t_{i_0} + \beta_{i_0} + \tau_{i_0} B w) \geq \max_{i=0}^{p-1} (G_i B t_i + \beta_i + \tau_i B w)$.

The start up for processor i is $T_s = \sum_{j=0}^{i-1} (G_j B t_j + \beta_j + \tau_j B w)$ and the startup time between bands becomes $T'_s = \sum_{i=0}^{p-1} (G_i B t_i + \beta_i + \tau_i B w)$

The computation time for the last processor is

$$T_c = \begin{cases} \frac{M}{B} G_i B t_{p-1} & \text{if } G_i B t_{p-1} \geq \max_{i=0}^{p-2} (G_i B t_i + \beta_i + \tau_i B w) \\ \frac{M}{B} G_i B t_{p-1} + (\frac{M}{B} - 1)(G_{i_0} B t_{i_0} - G_{p-1} B t_{p-1} + \beta_{i_0} + \tau_{i_0} B w) & \text{otherwise.} \end{cases}$$

Table 3. Accuracy of the Model: Block Cyclic Mapping with Buffering. $G_F = 4; G_S = 2$

| p=4 - n=60 - B=10 - M=500 | | | | p=6 - n=90 - B=10 - M=500 | | | |
|---------------------------|-----------|---------|-----------|---------------------------|-----------|---------|-----------|
| Config. | Real Time | Model | Error | Config. | Real Time | Model | Error |
| F S F S | 808.576 | 734.922 | -9.11E-02 | F S F S F S | 824.217 | 739.633 | -1.03E-01 |
| S F S F | 806.595 | 734.738 | -8.91E-02 | S F S F S F | 806.218 | 740.365 | -8.17E-02 |
| F F S S | 822.616 | 734.738 | -1.07E-01 | F F F S S S | 824.244 | 739.633 | -1.03E-01 |
| S S F F | 806.581 | 734.737 | -8.91E-02 | S S S F F F | 806.192 | 740.364 | -8.17E-02 |

Table 4. Accuracy of the Model: Block Cyclic Mapping with Buffering

| p=8 - n=120 - $G_F = 4; G_S = 2$ - B=10 - M=500 | | | |
|---|-----------|---------|-----------|
| Configuration | Real Time | Model | Error |
| F S F S F S F S | 825.287 | 746.178 | -9.59E-02 |
| S F S F S F S F | 815.530 | 745.995 | -8.53E-02 |
| F F F F S S S S | 828.470 | 746.178 | -9.93E-02 |
| S S S S F F F F | 812.473 | 745.995 | -8.18E-02 |

and the computation time for the last processor in an inner band becomes

$$T'_c = M(G_{p-1}Bt_{p-1} + \beta_{p-1} + \tau_{p-1}Bw + G_{i_0}Bt_{i_0} + \beta_{i_0} + \tau_{i_0}Bw - (G_{p-1}t_{p-1} + \beta_{p-1} + \tau_{p-1}Bw)).$$

The number of bands in this case turns to $Numbands = \frac{n'}{p}$

Considering T_s , T_c , T'_s and T'_c defined in terms of G_iBt_i , if $p = n'$ a pure block mapping is obtained and from proposition 1 the running time is $T_s + T_c$.

If $p < n'$ the total running time on a block cyclic mapping with n' stages and p processors can be directly obtained from theorem 2

$$T = \begin{cases} T_1 = T_s + (NumBands - 1)T'_c + T_c & \text{if } T'_c > T'_s \\ T_2 = (NumBands - 1)T'_s + T_s + T_c & \text{if } T'_s > T'_c \end{cases}$$

Tables 3 and 4 collects the results for the block cyclic mapping. To keep the number of bands per pipeline constant $\frac{n}{SlowProcs \times G_S + FastProcs \times G_F} = 5$ Although the errors increase up to 10%, we have the same invariances observed for the two former tables.

6 Concluding Remarks and Future Work

We have proposed a model for a large class of pipeline algorithms that considers heterogeneity due both to communication and computation. The analytical formula delivered by the model has been tested over a cluster of heterogeneous PCs using Linux. The computational results prove the effectiveness of the approach

and the accuracy of the predictions. There are several challenges and open questions to approach in the near future. The tuning problem for a pipeline over an heterogeneous platform is still an open question, i. e., to determine the optimal grain, buffer size and number of processors for a given problem and architecture. The predictive capability of the proposed model constitutes the basis for the study of this problem.

References

- [1] R. Andonov and S. Rajopadhye. Optimal Orthogonal Tiling of 2-D Iterations. *Journal of Parallel and Distributed Computing*, 45:159–165, September 1997. 442, 443
- [2] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. K. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *International Workshop on Heterogeneous Computing (HCW'99), in conjunction with IPPS'99*, pages 159–165, April 1999. 441
- [3] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. *Rapport de recherche de l'INRIA- Rhone-Alpes*, RR-4156:21 pages, April 2001. 441
- [4] L. M. Moreno, F. Almeida, D. Gonzalez, and C. Rodriguez. The Tuning Problem of Pipelines. In *EuroPar*. Springer-Verlag, 2001. 442, 443

Architectures for an Efficient Application Execution in a Collection of HNOWS

A. Furtado¹, A. Rebouças¹, J. R. de Souza², D. Rexachs¹, and E. Luque¹

¹ Arquitectura de Ordenadores y Sistemas Operativos, Dep. Informática
Universidad Autónoma de Barcelona, España
{adhvan, andre}@aows10.uab.es
{d.rexachs, e.luque}@cc.uab.es

² Curso de Informática, Universidade Católica do Salvador, Bahia, Brasil
josemar@ucsal.br

Abstract. To achieve high performance, an efficient task scheduling and load balance model is necessary, our goal is to optimize the execution of parallel algorithms over a Collection of geographically separated Heterogeneous Clusters of Workstations (CoHNOWS). Some architectures are tested and an adaptive Master/Worker model is proposed, where Cluster's workers are grouped by a sub-master that acts as a single point of control, and a separate machine is used to manage inter-cluster communications over WAN. Due to network latency unpredictability, an algorithm with dynamic data distribution is used, adapting each cluster's data load. The chosen benchmark algorithm was the Matrix Multiplication, because of its scalability facilities, and the MPI communications library for its portability. The testbed system used is composed of two heterogeneous dedicated clusters (HNOW), one in Spain and the other in Brazil, using a non-dedicated WAN (Internet) as the interconnection network between them.

1 Introduction

Among distributed computing platforms, heterogeneous networks of workstations are an efficient and cheap solution for data intensive computation. Many research centers have already taken advantage of their LAN facilities to build local clusters for parallel processing. With the spread of Internet use and its increasing bandwidth and reliability enhancement, e.g. Internet 2, the possibility to interconnect geographically scattered clusters became real; thus, collection of clusters can cooperate in solving a common large-scale problem. However efficiently managing such a heterogeneous environment is no trivial matter.

Load balance and data distribution, common problems in single HNOWS, become even more difficult when we group them to form Collections. This Collection of Clusters, made up of nodes, or clusters, each being a HNOW itself, may be

geographically scattered, and their interconnection network latency may not have a constant value [1]. Low bandwidth and large and variable latency represent important bottlenecks for execution time.

From a certain point of view, the idea of CoHNEWS is tied to the Grid concept of a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities [2].

We aim to provide an efficient load balancing and task scheduling methodology capable of improving the performance of algorithms on a collection of clusters by minimizing communication overheads and maintaining a dynamic control over the Mapping, Allocation and Data Distribution among the nodes. This paper analyzes the pros and cons of certain inter-cluster architectures. It is part of an ongoing study and represents our effort to efficiently execute algorithms in a Collection of Heterogeneous Network of Workstations.

Before testing models to interconnect clusters, an experimentation phase is necessary to characterize the environment. We realized that communications between clusters represent an important percentage of total computation time. Since we are dealing with an unpredictable communication layer, depending on the latency, the execution may even be interrupted. To extend the availability of the cluster, it is necessary to constantly control network usability and to ensure a dynamic rearrangement of the cluster's workload. This led us to use a SPMD, Master-Worker programming paradigm. We took advantage of this paradigm to approach a number of hierarchical models in order to gather data and optimize distribution over sub-clusters.

A relevant aspect that must be observed in any implemented architecture is the need to check the accessibility and performance of the workers at any time. This extra overhead caused by monitoring is compensated by a more extensive utilization of the machines and a consequent overall improvement in performance.

We analyzed the pitfalls in the execution of the parallel algorithm for this Collection of Heterogeneous Clusters in three distinct architectures: a single master sending tasks to isolated distributed workers; a master sending tasks to its local worker and to a sub-master in a different cluster; and finally, a master sending tasks to its local workers and to a sub-master in a different cluster but with a specialized computer, in the master's side, dedicated to managing communications between clusters. With the result empirically obtained by these three architectures, we propose an adaptive model to interconnect clusters.

Communication between cluster nodes and machines in the cluster is carried out by message passing; for this task, we use the standard MPI communication library [3], namely, the MPICH [4] implementation.

We chose the Matrix Multiplication (MM) algorithm as our initial benchmark application because it is highly scalable and its workload can be easily modified, thus being well suited to our research. The MM problem with different-speed processors turns out to be surprisingly difficult. Some research has been done in this field; a study by the LIP group [5] has proved the NP-completeness of the MM problem over heterogeneous platforms. A new parallel MM algorithm was proposed by [6] in order to take advantage of the specific characteristics of workstation networks, this work proves that MM problems executed in Heterogeneous NOWS spend relevant time with communications, even in an algorithm designed specifically for HNOWS, real

communication performance achieved is poor and is directly translated to the total running time.

The following sections present our study in further detail. Section 2 describes MM algorithms on Parallel Machines. The environment characterization is described in Section 3. The different architectures used and the experimental work undertaken are shown in Section 4, and finally, Conclusions and further work are presented in Section 5.

2 Benchmark Algorithm

The MM Algorithm is a simple but important Linear Algebra Kernel, and acts as a standard benchmark. Additionally, there are several distributed MM algorithms available for Clusters of Workstations [7]. Finally, it is highly scalable and its workload can easily be modified. MM is an $O(n^3)$ algorithm, so computation of a large matrix can spend considerable time in common machines.

Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies. Instead of operating on entire rows or columns of an array, blocked algorithms operate on sub matrices or blocks, so that data loaded into the faster levels of the memory hierarchy are reused [8]. Most of the reported parallel MM algorithms are blocked-like algorithms designed for 2D Mesh or Torus interconnected multicomputers. These algorithms can be classified as “Broadcast and Shift”, based on the Fox algorithm [9] or “align and shift”, based upon the Cannon work [10]. We assume that A, B and C are identically decomposed into blocks (Fig. 1.)

$$\begin{array}{cccc} A_s^{00} & A_s^{01} & A_s^{02} & A_s^{03} \\ A_s^{10} & A_s^{11} & A_s^{12} & A_s^{13} \\ A_s^{20} & A_s^{21} & A_s^{22} & A_s^{23} \\ A_s^{30} & A_s^{31} & A_s^{32} & A_s^{33} \end{array}$$

Fig. 1. The square block decomposition of matrix A, onto a 4x4-blocked matrix. The “s” refers to sub blocks

2.1 An Master-Worker Version of the Blocked MM Algorithm

In this version, the Master dynamically distributes the blocks of the matrices A and B to workers. They multiply received blocks (sub-matrices) using a serial MM algorithm and return the result to the Master, where it will be accumulated.

The algorithm itself works by sending an entire row of B blocks for each A block. We define A and B as being of n order, n' being the order of the blocked matrix (A_s , B_s and C_s). Initially, it reads the first A block and sends it together with all B's first row blocks. Once we had reached the last block of B's first row, we then read the second A block and the first block of B's second row, and so on. The algorithm ends

when we have sent the last block of A and the last block of B. Each block of C can be represented by the equation below (1):

$$C_s^{lk} = \sum_Q A_s^{lQ} * B_s^{Qk} \quad (1)$$

This algorithm, through the master, has the ability of dealing with large matrix sizes by using the disc capacity. So far, the master has the task of determining the matrix and the block sizes. The workers will deal only with the space available in the main memory, which limits our block size to the capacity of the poorest machine's main memory size. For this reason, we still need to characterize all machines being used.

An important aspect here is the blocks' arrival order. The master sends blocks dynamically to the workers, and there is no way to ensure the order in which they arrive. It is therefore necessary to know from what part of C the block belong to. In order to later recognize those blocks, the master sends the respective positions from A and B, and also keeps a structure which maps all pairs of blocks that are required to build each block of C.

As one C block arrives, the master stores it into an auxiliary file. Another structure is responsible for mapping the position of C, which each stored block belongs to. As there are no further C blocks to receive, the master loads them, one by one, from the auxiliary file. The loading process is not arbitrary: the master loads and sums the block according to the C block it wants to calculate. Once the C block is assembled, the master stores it into the matrix C file.

3 Environment Characterization

In this section, we will present the clusters that we work with, outline the methodology followed and show the results we have obtained by independently working with each cluster. To correctly design efficient architectures we must bear in mind what kind of problems we might expect in this environment. Most particularly, network and cluster computational power must be analyzed.

The environment is composed of two dedicated heterogeneous low-cost computer clusters (HNOWS), using a non-dedicated WAN (Internet) as the interconnection network between them, and a dedicated LAN (Ethernet 10Mb/s) to interconnect the machines inside both clusters. All machines works under Linux Red Hat (7.0 and 7.2), and we use the MPICH (version 1.2.2.1) distribution of MPI. The machines are physically allocated in the UCSal, Salvador, Bahia, Brazil, and the UAB, Bellaterra, Barcelona, Spain.

Our situation corresponds to a highly unpredictable, since we are using a public WAN to connect them. The M/W paradigm itself facilitates centralization of the system management. We use the idea of a central point that should function as a Global Master, managing all communications with others clusters, monitoring network behavior, controlling task execution and (ideally) also reallocating data, when necessary. Our experience in this kind of environment indicates that security is

an important aspect. At the very least, an extra layer of security must be implemented in a public WAN, such as the Internet.

We characterize each machine by using a serial matrix multiplication algorithm. This is important in order to determine how much work must be distributed to each machine and to assess the best data granularity. Using different matrix sizes, we measure the execution time in each cluster's machine and determined their performance (Tab. 1).

After defining the environment and characterizing the cluster's machines, we used a dynamic parallel version of the described MM algorithm to measure local cluster performance (Fig. 2).

Table 1. MFLOPS for each cluster machine for different matrix sizes of the serial MM algorithm

| Machines | Matrix Order (nxn) | | | | | | |
|--------------------------------|--------------------|------|------|------|------|------|------|
| | 100 | 400 | 500 | 800 | 1000 | 1500 | 2000 |
| Pentium 166 Mhz (UCSal) | 1,52 | 1,26 | 1,24 | 1,14 | 1,1 | 0,91 | 0,05 |
| Pentium III 500 Mhz (UCSal) | 12,05 | 7,93 | 7,51 | 7,45 | 6,57 | 6,41 | 6,3 |
| Pentium Celeron 433 Mhz (UAB) | 14,29 | 9,1 | 8,34 | 8,12 | 7,65 | 7,08 | 3,77 |
| Pentium II 200Mhz (32Mb) (UAB) | 1,77 | 1,42 | 1,5 | 1,45 | 1,28 | 1,22 | 1,24 |
| Pentium II 200Mhz (64Mb) (UAB) | 2,02 | 1,61 | 1,52 | 1,45 | 1,29 | 1,22 | 1,25 |

As the network plays an important role in our system, some characterization was also needed once we had to determine packet size, in addition to some form of network control. The characterization using system primitives (PING-PONG) and basic MPI's communication functions (MPI_Send / MPI_Recv) showed us that with large packets we can collect extremely different results simply by performing the same test at different moments. Due to the network behavior with MPI (Fig. 3) we chose blocks of 100*100 size.

We drew certain important conclusions from these experiments: first, that we needed an algorithm that dynamically distributes data, adaptive enough to masquerade the network's unpredictable latencies. Also, that data distribution should observe memory hierarchy in each machine. The correct data granularity should be one that takes advantage of each machine's main memory without turning the slowest machine into the system's bottleneck.

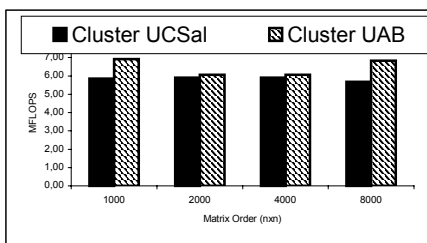


Fig. 2. Performance of the parallel MM algorithm

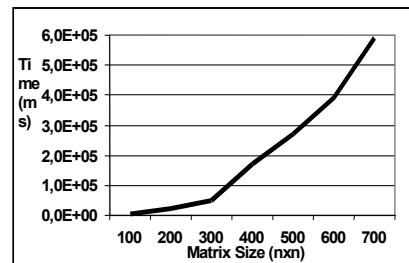


Fig. 3. MPI latency

4 CoHNEWS Architectures

After the whole phase of characterization, we began to test the cooperation between both clusters. We worked with three different architectures, in order to compare them and to assess their contribution. To analyze these architectures, we used two tests: the first one dividing a FLOAT data type 800*800 Matrix in blocks of 10*10, and other one using 1000x1000 FLOAT data type matrix divided into sub matrices of size 100x100.

The previous cluster and network characterization showed us the important drawbacks of the environment: the high latency, low bandwidth and unpredictable behavior of the interconnection network. These characteristics are potential bottlenecks for an efficient execution of the system. We aimed to design different architectures with the available components in order to minimize the effects of the problems outlined above. The first and direct architecture that we contemplated consisted of connecting distant machines as common workers, thus neglecting the type of connection between them (Fig.4). We assumed that we had 9 machines: 1 master and 8 workers directly connected to the master. Since there is no central controller, the control's signal between master and distant workers would have traffic in the slow link, thus slowing down the Master's response to the local worker's requests.

Due to MPI communications problems caused by heavy traffic in the inter cluster network, it was impossible to issue tests with large packets. Even with small packets, the execution times of figure 4 show that simply adding workers from a distant cluster is not an efficient approach, since the large network latency causes poorer execution time for the grouped Cluster than for one of the isolated clusters.

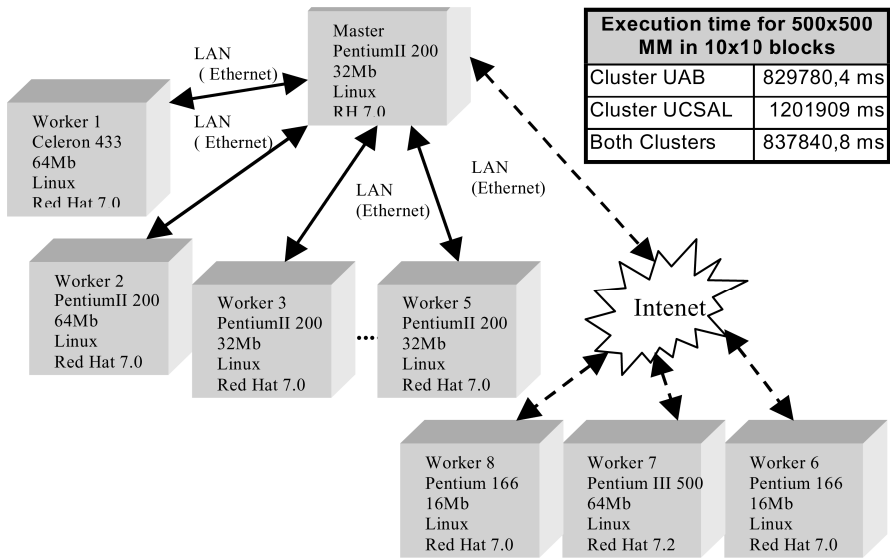


Fig. 4. Regular Master/Worker architecture – The Master, W1, 2, 3, 4 and 5 located in Spain. W6, 7 and 8 located in Brazil

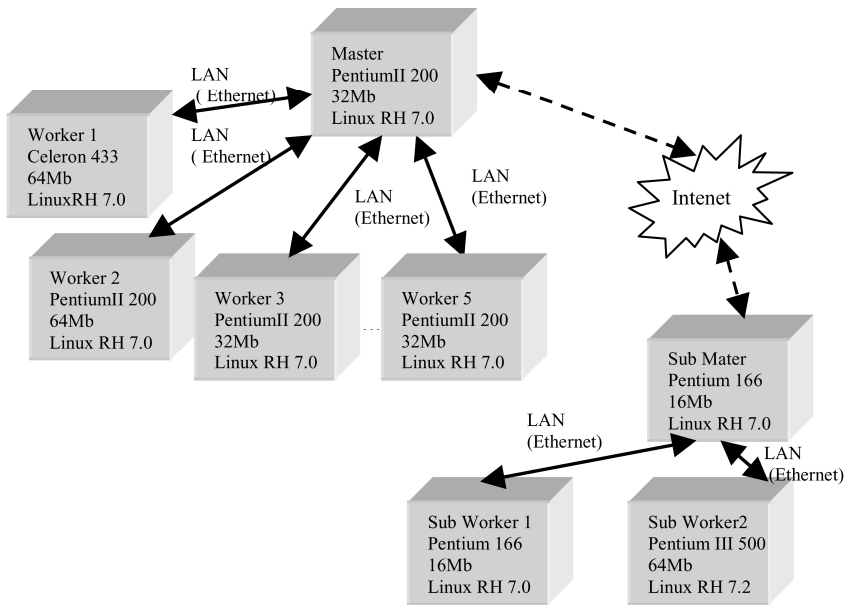


Fig. 5. Sub-Master architecture – The Master (located in Spain) exchanges messages with only one distant machine (Sub-Master – Located in Brazil)

The second approach (Fig. 5) introduced the idea of a Sub-Master to centralize the distant workers' incoming communication and to enable control of the task distribution among them. This architecture centralizes communications between Brazil and Spain: instead of having multiple points of communication between those machines, the sub-master in Brazil plays the role of a gateway, as a single point of communication between the clusters. It is responsible for receiving the packets and re-distributing them among local workers (sub-workers). The Master has the task of communicating with its local workers and with the remote Sub-Master. The Master and the Sub-Master should centralize all distant communications, but should also reduce the overhead related to the management of more than one distant connection.

Comparing two 1000*1000 MM algorithm executions in this model, ran on different days, we observe how the network can influence execution, and how adaptive is this algorithm with dynamic data distribution (Tab. 2a).

The unpredictable behavior of our particular interconnection network (WAN) is reflected in the data above. In the first execution, we assigned 279 blocks to the fastest worker and 59 (30 to sub-worker 2 and 29 to sub-worker 1) to the remote cluster. In the second sample, we assigned 370 blocks to the fastest worker and 26 (13 to sub-worker 1 and 13 to sub-worker 2) to the remote cluster. In theory, the best execution time should be associated with the second sample, because we took fuller advantage of the fastest worker and reduced the usage of the slow link. The fact is that the network connection made the difference, because at that specific moment the costs associated with the transmission of 26 blocks to the remote cluster were far greater than those associated with the first sample, when we sent 59 blocks. As the "receive"

is a blocking operation, when the master receives one packet from the Sub-Master it will block until the packet has arrived. The time spent receiving one packet from Brazil is longer than that spent receiving packets from Spain. Thus, local workers still loose CPU time whilst waiting for the master to complete these distant-reception operations. It is easy to observe the benefits of uncoupling slow link communications from the master by introducing a third architecture in which a specialized worker is responsible for managing communications between clusters (Fig. 6).

Notice that the local Spanish cluster now has one effective work machine less than in the previous architecture; nevertheless the MFLOPS obtained are almost 4 times those achieved in the other configuration (Table 2b). We verified that the time the master was loosing through sending and receiving information in the slow link interfered in its capabilities to attend to local workers' requests. Using this architecture, local machines are better exploited and the workload is better distributed throughout both clusters.

Table 2. Sub-Master and Communication Manager Execution Comparison. Worker 1 – Celeron 433 MHz, Worker 2, 3,4 and 5 – Pentium II 200 MHz, Sub-Worker 1 – Pentium 166 MHz, Sub-Worker 2 – Pentium III 500 MHz. (a) Sub-Master Execution in two different days. (b) Execution with communication Manager

| | Sub-Master (a) | | Communication Manager (b) |
|----------------|------------------------|---------------|---------------------------|
| | 14/05–11:00hs | 15/05–12:00hs | |
| Machines | Total Amount of Blocks | | Total Amount of Blocks |
| Worker 1 | 279 | 370 | 412 |
| Worker 2 | 166 | 153 | 196 |
| Worker 3 | 165 | 151 | 191 |
| Worker 4 | 166 | 152 | 190 |
| Worker 5 | 165 | 148 | 0 (Com. Manager) |
| Sub-Worker 1 | 29 | 13 | 5 |
| Sub-Worker 2 | 30 | 13 | 6 |
| | | | |
| Exec Time (ms) | 641.889,6 | 645.403,8 | 185.163,30 |
| MFLOPS | 1,56 | 1,55 | 5,40 |

Finally, we moved the Global Master to Brazil and performed certain tests using the Brazilian cluster as the principal. We used the Communication Manager architecture and performed a number of tests using different matrix sizes. Table 3 shows the experiment results, with one machine as the master and other dedicated to communication, only one remains as local worker, thus most of the problem must be computed in the distant workers; due to the large latency of the network, this is not totally useful. In an optimum CoHNOW architecture, no difference should be observed when changing master location.

With the support provided by evaluating the distinct architectures implemented, and once we had realized the need to group workers and separate slow inter-cluster communication from the master, a suitable architecture was then capable of being proposed for this problem.

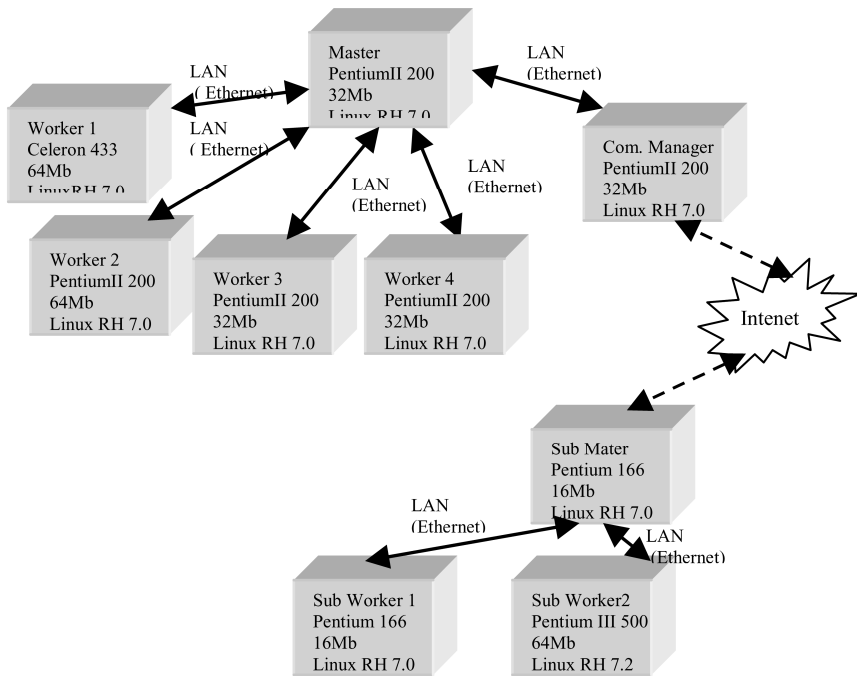


Fig. 6. Communicator Manager architecture – The Master in Spain exchanges messages only with local machines. A Communication Manager is responsible for distant communications

The design is very similar to the latter, but the Sub-Master can now buffer data, creating the possibility of overlapping computation with communication in the sub-cluster. The idea is to maintain a constant data flow to the far cluster. The sub-workers are uncoupled from the slow link, since the Sub-Master also acts as a communication manager.

Some data replication is necessary to ensure the adaptability of the system. Data is sent to the Sub-Master in an inverse order than that sent to local workers; therefore, if a given machine from the local cluster finishes its job, the last job that was assigned to the distant Cluster is then allocated to it.

The first one to finish the task will provide the correct data, and subsequent data will be invalidated. This represents an automatic load balance, when the inter-cluster network has a good bandwidth, the Sub-Cluster will contribute more to the final task; when bandwidth is poor, the Sub-Cluster will contribute less. Note that, in both cases, the work done by the distant Cluster is the same.

Table 3. CoHNOWS results with the Global Master in Brazil.

| | Spain to Brazil | | Brazil to Spain | |
|----------------|-----------------|---------|-----------------|---------|
| | 1000 | 2000 | 1000 | 2000 |
| Exec Time (ms) | 185163,3 | 1413924 | 238750,6 | 1688802 |
| MFLOPS | 5,40 | 5,65 | 4,18 | 4,73 |

5 Conclusions

Work with heterogeneous clusters demands a characterization of the system's computational power and of the cost of communications between these clusters. Communication times in CoHNEWS applications are an important ratio of the total computation time; the use of an unpredictable and relatively slow inter-cluster network connection such as the Internet demands a more intensive control of communication availability.

The Master-Worker paradigm provides a single point of control, ruling task execution and supervising workers' performance. The programs must allow changes in the load distribution, a key issue in any heterogeneous systems.

Static allocation can lead to better results within a stable environment running a well-known program, but previous characterization is necessary. In an unpredictable environment where communication and execution times are variable, static allocation gives poor performance and the dynamic allocation approach is therefore more suitable, when carefully analyses of the data granularity is done.

In order to efficiently execute algorithms in CoHNEWS, further work is required such as defining optimum block size, designing protocols and dynamic management strategies for data communication and implement scheduling policies that allow to overlap communication with computation are all part of this ongoing study.

References

- [1] Buyya, R.: High Performance Cluster Computing: Architectures and Systems, Vol 1. Prentice Hall PTR (1999).
- [2] Foster, I., Kesselman, C.: The Grid, Blueprint for a New Computing Infrastructure, pp. 15-51, Morgan-Kaufmann (1999).
- [3] Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. Scientific and Engineering Computation Series, Massachusetts Intitute of Technology (1999).
- [4] Gropp, W., Lusk, E., Doss, N., Skjellum A. "A high-performance, portable implementation of the {MPI} message passing interface standard", Parallel Computing. Vol. 22, no. 6, 789--828, sep 1996.
- [5] Beaumont, O., Rastello, F., Robert, Y.: Matrix Multiplication on Heterogeneous Platforms. IEEE Trans. On Parallel and Distributed Systems, Vol. 12, No. 10. (October 2001).
- [6] Tinetti, F., Quijano, A., Giusti, A., Luque, E.: Heterogeneous Network of Workstations and the Parallel Matrix Multiplication. Euro PVM/MPI 2001, Y. Cotronis and J. Dongarra, eds., pp. 296-303 (2001).
- [7] Wilkinson, B., Allen, M.: Parallel Programming Techniques and Applications Using Networking Workstations, Prentice-Hall inc. (1999).
- [8] Lam, M. S., Rothberg, E., Wolf, M. E.: The Cache Performance and Optimizations of Blocked Algorithms, Fourth Intern. Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto CA (April 1999).

- [9] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D.: Solving Problems on Concurrent Processors, Vol. 1, Prentice Hall, Englewood Cliffs New Jersey(1988).
- [10] Cannon, L. E.: A Cellular Computer to Implement the Kalman Filter Algorithm, Ph.D thesis, Montana State University, Bozman Montana (1969).

Author Index

| | | | |
|---------------------------|----------------|-------------------------------|------------|
| Albada, Dick van | 4 | Fuentes, A. | 33 |
| Almeida, F. | 440 | Furtado, A. | 449 |
| Astalos, Jan | 105 | Gallopoulos, Efstratios | 199 |
| Aszódi, András | 148 | García, Félix | 306 |
| Augustin, Werner | 271 | Geissler, Michael | 421 |
| Baliś, Bartosz | 41 | Geist, Al | 6 |
| Bekas, Constantine | 199 | Germain, Cecile | 323 |
| Belhadj-Aissa, A. | 138 | Gerndt, Michael | 11 |
| Bernholdt, David E. | 260 | Giné, Francesc | 156 |
| Bißeling, Georg | 401 | González, D. | 440 |
| Blaszczyk, Andreas | 70 | Gropp, William | 7, 12, 280 |
| Bosilca, George | 323 | Guibault, François | 243 |
| Brightwell, Ron | 331, 385 | Habala, Ondrej | 105 |
| Bubak, Marian | 14, 16, 41, 50 | Hernández, Porfidio | 156 |
| Bustos, Rafael | 368 | Hluchy, Ladislav | 105 |
| Calderón, Alejandro | 306 | Höfinger, Siegfried | 62, 148 |
| Cappello, Franck | 323 | Hoppe, Hans-Christian | 401 |
| Carretero, Jesús | 306 | Huang, Wei | 314 |
| Casquero, Ángel | 368 | Ibáñez, María B. | 226 |
| Causin, Paola | 78 | Imamura, Toshiyuki | 288 |
| Chapman, Barbara | 13 | Jorba, Josep | 368 |
| Chong, Kyun Rak | 191 | Juhasz, Zoltan | 8 |
| Coli, Moreno | 349 | Kabir, Yacine | 138 |
| Cotronis, Yiannis | 252 | Karl, Wolfgang | 114 |
| Cuevas, J. | 33 | Kasprzyk, Leszek | 122 |
| Czarnul, Pawel | 208 | Kim, Junghwan | 191 |
| Dias, Luís | 96 | Kitowski, Jacek | 25 |
| Dias, Miguel | 96 | Kohl, James A. | 260 |
| Dompierre, Julien | 243 | Kokiopoulou, Efrosini | 199 |
| Dongarra, Jack | 1 | Kotocova, Margareta | 234 |
| Dutka, Lukasz | 25 | Kranzlmüller, Dieter | 357 |
| Elwasif, Wael R. | 260 | Krawczyk, Henryk | 376 |
| Espenica, Roberto | 341 | Krysztop, Bartosz | 376 |
| Fahringer, Thomas | 421 | Lagier, Pierre | 401 |
| Fedak, Gilles | 323 | Lamberti, Fabrizio | 165 |
| Fernández, Javier | 306 | Latour, Jean | 401 |
| Freitas, Conceição | 96 | Lazzarino, Oscar | 165 |
| Froehlich, David | 105 | Levi, Paul | 174 |
| Funika, Włodzimierz | 41, 50 | | |

- Luque, Emilio156, 368, 449
 Lusk, Ewing9, 12
 Ma, Jie314
 Maccabe, Arthur B.331
 Macías, Elsa M.130
 Madsen, Georg421
 Malawski, Maciej16
 Malouin, Éric243
 Marco, J.33
 Marco, R.33
 Margalef, Tomàs368
 Margaritis, Konstantinos G. ...431
 Martínez-Rivero, C.33
 Medeiros, Pedro341
 Menéndez, R.33
 Michailidis, Panagiotis D.431
 Miglio, Edie78
 Miller, Barton P.10
 Moreno, L.M.440
 Nawrowski, Ryszard122
 Nguyen, Giang Thu234
 Nägeli, Hans-Heinrich88
 Pagourtzis, Aris217
 Palazzari, Paolo349
 Pérez, Jose M.306
 Piriya Kumar, Douglas Antony Louis
 174
 Plachetka, Tomas296
 Ponce, O.33
 Potapov, Igor217
 Rabenseifner, Rolf174, 410
 Rebouças, A.449
 Reussner, Ralf271
 Rexachs, D.449
 Riesen, Rolf331
 Rodríguez, C.440
 Rodríguez, D.33
 Rotaru, Tiberiu88
 Roy, Robert243
 Rughi, Rodolfo349
 Rytter, Wojciech217
 Sanna, Andrea165
 Schindler, Torsten148
 Schulz, Martin61, 114, 357
 Selikhov, Anton323
 Serio, Angela Di226
 Simo, Branislav105
 Simoncini, Valeria199
 Sloat, Peter4
 Solsona, Francesc156
 Song, Ha Yoon191
 Souza, J. R. de449
 Suárez, Alvaro130
 Sunderam, Vaidy3
 Supalov, Alexander401
 Szeplieniec, Tomasz41
 Takemiya, Hiroshi288
 Tischler, Florian183
 Tomczewski, Andrzej122
 Träff, Jesper Larsson392
 Tran, Viet Dinh105, 234
 Trinitis, Carsten61, 114
 Truong, Hong-Linh421
 Tsiatsoulis, Zacharias252
 Tsujita, Yuichi288
 Turała, Michał14
 Uhl, Andreas183
 Uhl, Axel70
 Wang, Zhe314
 Wismüller, Roland41, 50
 Worsch, Thomas271
 Yamagishi, Nobuhiro288
 Zajac, Katarzyna16
 Zunino, Claudio165