Matthias S. Müller
Bronis R. de Supinski
Barbara M. Chapman (Eds.)

Evolving OpenMP in an Age of Extreme Parallelism

5th International Workshop on OpenMP, IWOMP 2009
Dresden, Germany, June 3-5, 2009
Proceedings

Springer

Volume Editors

Matthias S. Müller
Technische Universität Dresden
Zentrum für Informationsdienste und Hochleistungsrechnen
01062 Dresden, Germany
E-mail: matthias.mueller@tu-dresden.de

Bronis R. de Supinski
Lawrence Livermore National Laboratory
Center for Applied Scientific Computing
Livermore, CA 94551-0808, USA
E-mail: bronis@llnl.gov

Barbara M. Chapman
University of Houston
Department of Computer Science
Houston, TX 77204-3475, USA
E-mail: chapman@cs.uh.edu

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.1, C.1, F.1, G.4, C.2.4, D.2.5, D.3.2, D.4.7

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-642-02284-7 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-02284-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12695836 06/3180 5 4 3 2 1 0

Preface

OpenMP is an application programming interface (API) that is widely accepted as a de facto standard for high-level shared-memory parallel programming. It is a portable, scalable programming model that provides a simple and flexible interface for developing shared-memory parallel applications in Fortran, C, and C++. Since its introduction in 1997, OpenMP has gained support from the majority of high-performance compiler and hardware vendors. Under the direction of the OpenMP Architecture Review Board (ARB), the OpenMP specification is undergoing further improvement. Active research in OpenMP compilers, runtime systems, tools, and environments continues to drive OpenMP evolution. To provide a forum for the dissemination and exchange of information about and experiences with OpenMP, the community of OpenMP researchers and developers in academia and industry is organized under cOMPunity (www.compunity.org). This organization has held workshops on OpenMP since 1999.

This book contains the proceedings of the 5th International Workshop on OpenMP held in Dresden in June 2009. With sessions on tools, benchmarks, applications, performance and runtime environments it covered all aspects of the current use of OpenMP. In addition, several contributions presented proposed extensions to OpenMP and evaluated reference implementations of those extensions. An invited talk provided the details on the latest specification development inside the Architecture Review Board. Together with the two keynotes about OpenMP on hardware accelerators and future generation processors it demonstrated that OpenMP is suitable for future generation systems.

OpenMP 3.0 has been adopted rapidly, as is evidenced by the number of available compilers and the use of new features by application developers. The OpenMP workshop series has made important contributions to the development of the specification and its adoption. This year's contributions clearly indicate that this role will continue in the future.

June 2009

Matthias S. Müller
Barbara M. Chapman
Bronis R. de Supinski

Organization

Committee of IWOMP 2009

General Chair

Matthias S. Müller	University of Dresden, Germany

Program Committee Chair

Matthias S. Müller	University of Dresden, Germany

Program Committee

Dieter an Mey	RWTH Aachen University, Germany
Eduard Ayguade	CIRI, UPC, Spain
Mark Bull	EPCC, University of Edinburgh, UK
Barbara Chapman	University of Oregon, USA
Bronis R. de Supinski	LLNL, USA
Rudolf Eigenmann	Purdue University, USA
Guang Gao	University of Delaware, USA
Rick Kufrin	NCSA, USA
Chunhua Liao	LLNL, USA
Federico Massaioli	CASPUR, Rome, Italy
Lawrence Meadows	Intel, USA
Mitsuhisa Sato	University of Tsukuba, Japan
Ruud van der Pas	Sun Microsystems, Geneva, Switzerland
Michael Wong	IBM, Canada

Steering Committee Chair

Bronis R. de Supinski	NNSA ASC, LLNL, USA

Steering Committee

Dieter an Mey	CCC, RWTH Aachen University, Germany
Eduard Ayguade	Barcelona Supercomputing Center (BSC), Spain
Mark Bull	EPCC, UK
Barbara Chapman	CEO of cOMPunity, Houston, USA
Rudolf Eigenmann	Purdue University, USA
Guang Gao	University of Delaware, USA
Ricky Kendall	ORNL, USA
Michaël Krajecki	University of Reims, France
Rick Kufrin	NCSA, USA
Federico Massaioli	CASPUR, Rome, Italy

Table of Contents

Parallel Simulation of Bevel Gear Cutting Processes with OpenMP Tasks

Paul Kapinos and Dieter an Mey

Center for Computing and Communication,
JARA, RWTH Aachen University, Germany
{kapinos,anmey}@rz.rwth-aachen.de

Abstract. Modeling of bevel gear cutting processes requires highly flexible data structures and algorithms. We compare OpenMP3.0 tasks with previously applied approaches like nesting parallel sections and stack based algorithms when parallelizing recursive procedures written in Fortran 95 working on binary tree structures.

Keywords: OpenMP 3.0, Tasking, Nesting, Fortran 90/95, Recursive Subroutines, Parallelization.

1 Introduction

Today manufacturers of bevel gears are confronted with continuously growing demands for cost effectiveness of the manufacturing process. This is also true for the cutting of bevel gears. To enhance the productivity of the manufacturing processes, the processing time has to be reduced. In order to further improve productivity, the tool life has to be maximized so that the tooling time and the tool costs are minimized. By increasing the cutting velocity, a reduction in the primary processing time is obtained. This results in an increased load on the cutting edge of the tool. In order to predict the tool life during these processes, a detailed analysis of the cutting process is required to evaluate the chip creation conditions. The Laboratory for Machine Tools and Production Engineering (WZL, [1]) of the RWTH Aachen University develops the program KegelSpan [2] [3] [4] to simulate and study the manufacturing process of bevel gears.

Shared-memory parallelization with OpenMP was taken into account during the development of KegelSpan early on. OpenMP was chosen because of the possibility to gradually adopt parallelism on parts of premature program versions without rewriting lots of code, and the ability to easily handle complicated data structures. The recursive routines, which work on binary trees, were detected to consume a substantial part of the runtime in initial versions of KegelSpan and have been investigated and parallelized in [5]. In this paper we expand this work taking the long-awaited OpenMP 3.0 tasking approach into consideration.

In section 2 we briefly describe the program KegelSpan [2]. In section 3 we discuss the parallelization strategies which we applied to the recursive routines, which have been selected for parallelization after profiling the whole program. We present and discuss our performance measurements of the parallelized routines in section 4 and conclude in section 5.

2 Simulation of Bevel Gear Cutting Processes

The program KEGELSPAN developed by the WZL [1] simulates bevel gear cutting with defined cutting edges. It is written in Fortran 95 heavily using many advanced language features like modules, recursive functions, Fortran pointers and operator overloading. In theory, the process of a single cutting process is well understood, but the production of a complete bevel gear consists of a series of cutting processes. By today's standards the tool wear, life time and critical wear behavior are determined experimentally. The target of the development of KEGELSPAN is to determine optimal process attributes (cutting speed, feed etc.) on the basis of calculated variables like chip thickness, unwinded cutting edge length and to better predict the tool wear behavior.

The tool's edges are represented by spacial lines. The kinetics of the movements of the cutting tool determines the movement of these lines in space and defines the hulls of the cutting edges.

These hulls, represented by a three-dimensional polygon grid, are examined in the order of the cutting edges for penetration with the polygon grid of the workpiece. If a penetration occurs, the difference volume, the so-called non-deformed chip, is computed by subtracting the surfaces of both polygon grids. Based on the non-deformed chip, the characteristic values can be determined. Whenever the individual cuts are computed, the polygon network of the workpiece is adaptively refined. This frequently leads to an unbalanced distribution of nodes and polygons in space.

In order to reduce the amount of computation by only examining neighboring parts of the polygon grids for mutual penetration, a binary space partitioning tree (BSP tree) is constructed. The polygons are recursively sorted into a hierarchy of nested axis aligned bounding boxes (AABBs), until the number of polygons per box falls below an adjustable threshold value. Because of the uneven distribution of the nodes in space (see figure 1 left) the binary tree (BSP Tree) tends to be quite unbalanced. However, an attempt to balance this tree according to the number of polygons per sub-box by setting the split plane through the mass center of all polygons turned out to be counter-productive. For geometrical and algorithmic reasons the program's runtime increased considerably.

The BSP trees are implemented with dynamic data structures and Fortran pointers in the KEGELSPAN program and recursive routines are used for tree traversal. As early performance analyses revealed that 75 percent of the total runtime was consumed in these recursive procedures, they were initially parallelized with "traditional" OpenMP approaches and now with OpenMP 3.0 tasking constructs as soon as compilers became available.

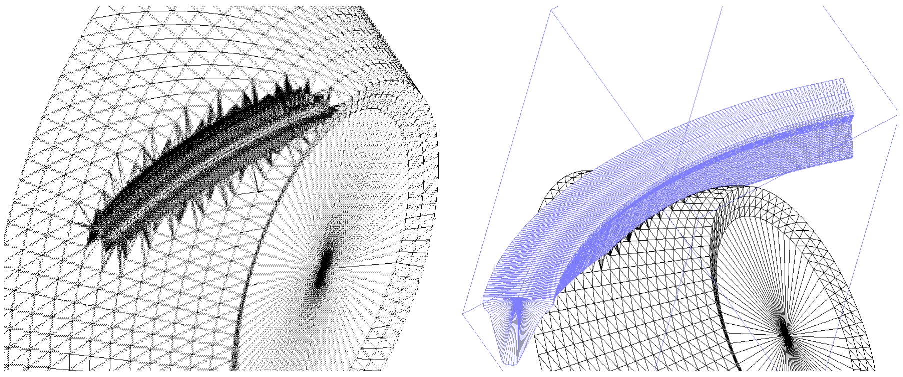


Fig. 1. Adaptive refinement (left) and penetration of meshes (right)

3 Parallelization of the Recursive Routines

For simplicity we only discuss routines which directly call themselves twice and are used for the traversal of binary trees, if these two evocations can safely be computed in parallel.

```
RECURSIVE SUBROUTINE BSPtraversal (tree)
TYPE (BSPtree) :: tree
.... do work here
CALL BSPtraversal (tree%left)
CALL BSPtraversal (tree%right)
END SUBROUTINE BSPtraversal
```

3.1 Parallelization with Nested `PARALLEL SECTIONS` Constructs

In the first approach we simply executed both recursive evocations in a `PARALLEL SECTIONS` constructs containing two sections:

```
RECURSIVE SUBROUTINE PrimerPar (tree)
TYPE (BSPtree) :: tree
.... do work here
!$OMP PARALLEL SECTIONS
!$OMP SECTION
  CALL PrimerPar (tree%left)
!$OMP SECTION
  CALL PrimerPar (tree%right)
!$OMP END PARALLEL SECTIONS  !   <---   barrier !
END SUBROUTINE PrimerPar
```

In order to activate more than 2 threads, nested parallelism has to be supported by the compiler and explicitly enabled. If there is no limitation of the nesting depth or the total number of threads, an additional thread will be forked for each node of the tree. We applied two different strategies to avoid overloading the system with thousands of threads which could easily happen in KEGELSPAN program.

Limiting the nesting depth to N also implies that the total number of threads is limited by 2^N. The computational load is distributed to the threads in a deterministic manner as the distribution is determined by the shape of the tree. If the subtrees differ in size, the tree is unbalanced and so is the work distribution. The OpenMP overhead is limited, as once the maximum nesting level is reached, the remaining subtree will be processed sequentially.

Limiting the total number of threads to M ("thread pool") also implies that the nesting level is limited by $M-1$. For each node the number of currently active threads is checked and as long as the limit is not reached, an additional thread is forked. This strategy potentially may lead to an automatic load balance. But, alas, the implied barrier at the end of the `PARALLEL SECTIONS` construct inserts additional overhead (see figure 2 left). Furthermore, the work distribution is non-deterministic such that even in the case of a balanced tree, a load imbalance may occur (see figure 2 right).

Up to version 2.5 of the OpenMP API there was no standard way to control the maximum nesting depth or the maximum number of threads of an OpenMP program. Although some compilers offered extensions to control the nesting behavior of applications we implemented this manually by using a variable which was decremented in every recursive call (limiting max. nested depth) or a shared variable containing the number of currently active threads (thread pool) in order to retain portability.

In OpenMP v3.0 new Internal Control Variables (ICV) allow to set the limits in a portable and easy way. The THREAD-LIMIT-VAR ICV may be controlled by the `OMP_THREAD_LIMIT` environment variable and is used to limit the size of the "thread pool". The MAX-ACTIVE-LEVELS-VAR ICV may be controlled by `OMP_MAX_ACTIVE_LEVELS` environment variable and is used to limit the maximum nesting depth.

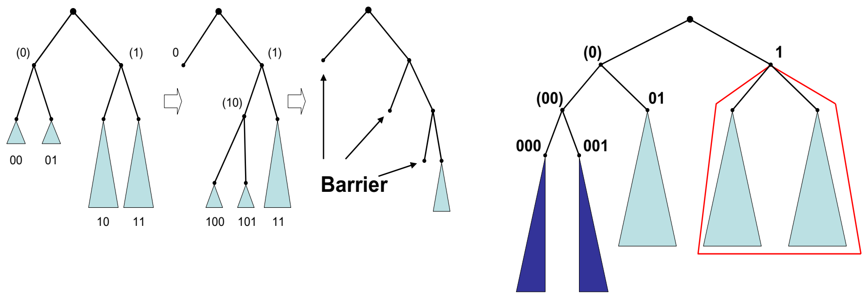


Fig. 2. Nested `PARALLEL SECTIONS` with 4 threads: The size of the triangles represents the amount of work in the subtrees.
Left: Limited self-balancing.
Right: Work scheduling is non-deterministic; bad scheduling on a balanced tree.

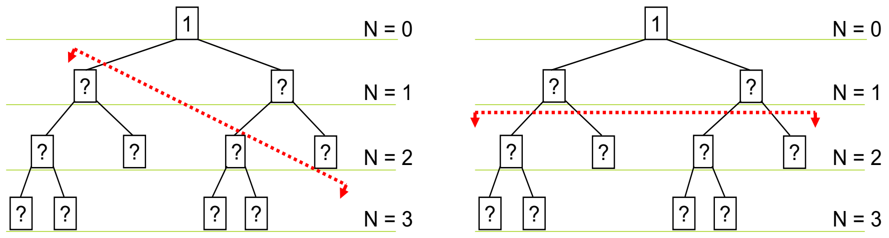


Fig. 3. Adaptive parallelization: Probability fronts for LIFO (left) or FIFO queue

3.2 Adaptive Parallelization

In an alternative approach we reimplemented the recursive procedures by an iterative algorithm. A compiler would similarly map the recursion to a stack based algorithm. But only the manual approach provides an opportunity for parallelization. A dynamic data structure (LIFO- or FIFO queue) is introduced to dynamically store the arguments of the previously recursive procedure call. The initial parameters are stored into this structure in an initialization phase and a team of threads executes a parallel loop until the queue is empty **and** no thread is working on any item any more. Accesses to the queue have to be guarded in critical regions.

The participating threads get an item from the queue, proceed with their work, and eventually put items into the queue - if the recursive pendant would have recursive evocations here.

This adaptive algorithm leads to an automatically balanced work load as long as the tree is not degenerated into a list.

```
SUBROUTINE BSPstack (tree)
TYPE (BSPtree)  :: tree
TYPE (Stack) :: myStack
.....
CALL StackPush (tree, myStack) ! Initialize
!$OMP PARALLEL
DO WHILE (queue is not empty .and. at least one thread is busy)
  CALL StackPop (localtemp, myStack)
  ... work on localtemp here
  IF ( node is not a leaf node )
    CALL StackPush (localtemp%left, myStack)
    CALL StackPush (localtemp%right, myStack)
  END IF
END DO
!$OMP END PARALLEL
.....
END SUBROUTINE BSPstack
```

As the queue has to be accessed by all threads in critical regions, it may become a bottleneck for a large number of threads and thus lead to a limited scalability. An implementation of a distributed queue may alleviate overhead.

The strategy of the queue has an impact on the processing sequence of the nodes.

A FIFO approach corresponds to a breadth-first traversal when executed serially, whereas the LIFO approach corresponds to a depth-first traversal. In parallel, there is no arrangement of node traversal enforced, but the trends still remain (see fig. 3)

The programming effort for the adaptive algorithm is considerably higher compared to the one employing nested `PARALLEL SECTIONS`, as it may impact the implementation of the work function as well due to the paradigm change from recursive to iterative.

3.3 Tasking

With OpenMP version 3.0 [6] the new tasking concept offers a thriving new alternative. A task is an enclosed computational assignment, which is defined first and executed later by any thread of the participating team. A task can even generate further tasks. After generating another task, the generating task can continue its execution first, while the generated task can be executed by any thread of the participating team immediately or later. At an (implicit or explicit) `TASKWAIT` barrier all previously generated tasks have to be completed.

Obviously, the tasking concept lends itself to the parallelization of recursive procedures by simply putting the recursive procedure evocation into task constructs.

A team of threads is forked and the master thread generates the first task to operate on the root node. Tasks which then are generated recursively, are equally executed by all threads of the team.

Here the `TASKWAIT` directive hardly affects performance, because threads arriving at this barrier can happily suspend execution and work on another task without any need to idle. This property of OpenMP tasks leads to an improved load balancing almost "for free".

```
.....
!$OMP PARALLEL
IF (master) THEN CALL BSPtask (root) ! Start, one Thread only
!$OMP END PARALLEL
.....
RECURSIVE SUBROUTINE BSPtask (tree)
TYPE (BSPtree) :: tree
.... do work here
!$OMP TASK
  CALL BSPpar (tree%left)
!$OMP END TASK
!$OMP TASK
  CALL BSPpar (tree%right)
!$OMP END TASK
!$OMP TASKWAIT ! <---   does not really affect performance !
END SUBROUTINE BSPtask
```

Usually a TASKWAIT barrier is necessary, if the output of a subtask is needed in the computation after the TASKWAIT directive. If this is not the case, as e.g. the subtask's output is stored in disjoint partitions of a shared memory region, there is the opportunity to save system resources because of a lower number of simultaneously active tasks. But it turns out that Fortran is at a disadvantage because Fortran pointers cannot be declared FIRSTPRIVATE on a TASK directive (see [6] section 2.9.3.4 on page 94) and without a TASKWAIT barrier, the variables' lifetime which are shared in the generating task might prematurely end (see [6] section 2.7 on page 61).

Additionally creating a task per tree node may lead to a slow-down due to the overhead of managing the tasks by the runtime system. A possible solution to this may be obtained by increasing the amount of work performed per task.

The merging of the work may be achieved by forbidding the spawning of additional tasks if an threshold nesting depth of tasks is reached. This may be implemented by an IF clause in the task definition or by a conventional if branch and calling the serial version of the routine. Furthermore, the call tree may be analyzed in more detail to handle tiny and fat nodes differently. The fat nodes can be sensibly parallelized by spawning an additional task, whereas tiny ones should be executed immediately.

4 Performance Experiments

Two recursive subroutines of KegelSpan have been parallelized with all the approaches described in the previous section. We compare programming effort and report on runtime measurements on two different platforms.

4.1 Work Distribution and Characteristics of the BSP Trees

Because of algorithmic reasons (adaptive refinement of meshes and unequal distribution of points in the space) the BSP trees in KegelSpan are heavily unbalanced. Furthermore, the shape and the growth of the trees varies a lot during the execution of the program and are specific to the particular use case. Hence, only an averaged overview of the tree growth and shape may be given which is valid only for the test data set.

Accumulated over the whole computation, there are about 230 thousands of leaves and almost the same number of inner nodes in the BSP trees. The average leaf depth is 12, with a lot of leaves (63 %) concentrated in the depth of 10 to 13. There are leaves in the depth from 6 to 20, too. On average, one subtree from the root node has about double the number of nodes than the other subtree, confirming that the trees are unbalanced.

The first of the parallelized routines, denoted as A, is an adopted variation of the well-known QuickSort algorithm. The amount of work decreases in every step and consequently the scalability of any parallelization of this algorithm is limited.

Table 1. Three approaches in comparison

	SECTIONS	Adaptive parallelization	Tasking
time to adopt, est.	1 hour	1 week	1 hour + “verification”
code lines added	less than 50	up to 1000	less than 50
work on unbalanced trees	bad	good	good
algorithmical changes?	no	maybe	no

In the second parallelized routine, denoted as B, almost whole work is concentrated in the leaves of the BSP tree. The work of inner nodes consists only of some reduction to merge the results of recursive calls.

The execution time of an average task is in the order of milliseconds, or millions of CPU cycles.

4.2 Programming Effort

We try to summarize and categorize our experiences in implementing the different parallelization approaches in table 1.

Parallelization with a nested `PARALLEL SECTIONS` construct was the simplest, because it is a simple and matured concept and it requires only few additional lines of code and no algorithmic changes.

The implementation of the adaptive parallelization algorithm required a complete rewrite of the related routines including some changes in the algorithm. Programming efforts and error-proneness were clearly higher.

The time to implement the tasking concept was comparable to the `SECTIONS` solution. But anyhow it took some time to verify the solution and to get the data scopes all right. The particular definition of default data scopes for the `TASK` construct compared to the `PARALLEL` construct (see [6] section 2.9.1.1 on page 79) leads to some confusion in determining the data scope used by default. Scoping seems to be a typical source of errors and we hope that the known data race detection tools will be ready to check task constructs soon.

The new runtime routines and the environment variables for the limitation of the nesting depth or the size of the thread pool provided by OpenMP 3.0 furthermore reduce the necessary programming effort. But they turned out to be less flexible than our hand-coded routines.

4.3 Speedup

We ran all variants of the parallelized routines on two different machines:

– a Fujitsu-Siemens RX200 S4/X equipped with two Intel Xeon E5450 quad core processors at 3.0GHz running Linux (denoted as "**Harpertown**" in the following) and
– a Sun T5120 equipped with one eight-core Sun Niagara T2 processor with 64 simultaneous threads at 1.4 GHz running Solaris (denoted as "**Niagara2**").

Table 2. Speedup compared to serial version, without thread binding. SD = SECTIONS with limiting the nesting depth, ST = SECTIONS with limiting the number of active threads (threadpool) , AF = adaptive parallelization with FIFO queue, AL = adaptive parallelization with LIFO queue (stack), T = tasking (without limits to task spawning). SD and ST employ our hand-coded versions to control the nesting behavior and not the new OpenMP 3.30 runtime functions.

		Niagara2					Harpertown									
		SUN Studio Express 07/2008 (**-fast**)										Intel 11.1.016 beta (**-O3**)				
	#Thr	SD	ST	AF	AL	T	SD	ST	AF	AL	T	SD	ST	AF	AL	T
A: buildBSPTree	1	0.98	1.02	0.99	0.97	1.00	1.00	1.00	0.96	0.97	0.99	0.93	0.93	0.91	0.93	1.00
	2	1.45	1.45	1.77	1.77	1.78	1.47	1.46	1.38	1.67	1.71	1.35	1.37	1.61	1.64	1.81
	4	2.31	1.97	2.72	2.76	2.78	2.30	1.94	2.52	2.61	2.53	2.17	1.82	2.45	2.48	2.83
	8	2.64	2.34	3.39	3.39	3.47	2.53	2.20	3.02	**3.09**	3.06	2.41	2.37	2.89	2.93	**3.49**
	16	2.98	2.70	3.69	3.69	3.81	2.63	2.39	3.01	**3.09**	3.06					
	32	2.98	2.89	3.59	3.53	3.83										
	64	2.97	2.97	1.30	1.28	**3.86**										
B: boxLpSchlArr	1	1.01	0.98	1.12	1.12	0.53	0.95	0.95	0.99	1.00	0.51	0.99	0.99	0.94	0.94	0.49
	2	1.41	1.42	1.83	2.00	0.98	1.02	0.98	1.03	1.07	0.60	1.20	1.19	1.44	1.52	0.82
	4	1.70	1.90	2.61	3.62	1.89	1.00	0.99	0.91	0.88	0.51	1.36	1.44	1.65	**1.98**	1.10
	8	1.94	2.18	3.86	6.01	3.53	0.97	0.88	0.92	0.86	0.45	1.43	1.69	1.54	1.82	1.06
	16	2.93	2.19	5.39	**8.59**	5.98	0.91	0.77	0.90	0.84	0.45					
	32	2.85	1.70	6.57	8.30	7.96										
	64	2.78	1.56	4.74	4.87	8.01										

Table 3. Accumulated serial execution times of the recursive routines in seconds (corresponding to a speedup of 1.0 in table 2)

	Niagara2	Harpertown	
	SUN Studio Express 07/2008		Intel 11.0.034 beta
A: buildBSPTree	157	17.9	16.6
B: boxLpSchlArr	501	54.9	38.8

On both Niagara2 and Harpertown we used the Sun Studio Express compiler and turned on high optimization with **-fast** whereas on Harpertown we additionally used the Intel Beta compiler (11.1.016) for which we were able to use the optimization with **-O3**, only if we omit the **-g** debug flag.

We observe a maximum speedup of 3.49 with 8 threads on Harpertown and 8.59 with 16 threads on Niagara2 (see tables 2). Please note the difference in absolute runtime (table 3): Because of the higher clock cycle, the superscalar architecture, and the large caches the Harpertown outperforms the Niagara2 by about one order of magnitude with the serial code version.

As Niagara2 contains 2 instruction units per core, 16 threads frequently are a sweet-spot for multi-threaded applications which do not demand a high memory bandwidth. The Niagara2 typically shines when running memory intensive codes with many (up to 64) threads waiting for memory in parallel. Here we obtain

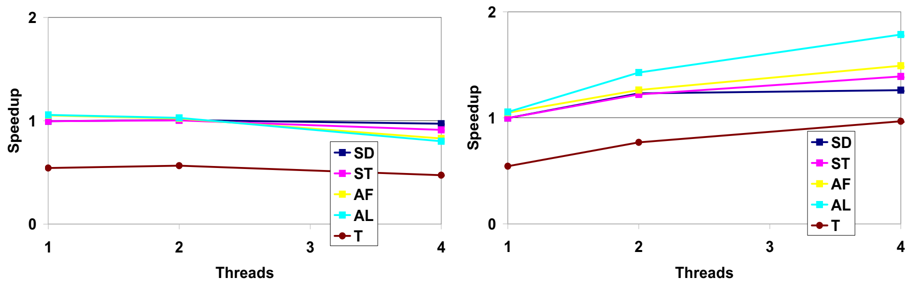


Fig. 4. Speedup on Dunnington with thread placement by **taskset**. Sun Studio compiler **-fast**.
Left: scatter - private cache (taskset -c 0,1,2,3)
Right: compact - shared cache (taskset -c 1,9,10,11,12,13)

a speedup of up to 3.86 with 64 threads on routine A and up to 8.59 with 16 threads on routine B. All program versions display a smooth behavior with respect to the number of threads employed, though the scalability of routine A is obviously limited.

On Harpertown, we frequently observed that 2 threads of a memory intensive code may consume the whole bandwidth of the front side bus while higher scalability can be observed for cache-friendly applications.

For KegelSpan a speedup of 3.49 (Intel compiler) or 3.09 (Sun Studio compiler) for the routine A with 8 threads is a tolerable result, in particular when taking the limited scalability of the QuickSort-like algorithm into account.

The parallel versions of routine B, that scaled up to 8.59 on Niagara2, do not perform well on Harpertown. The best speedup achieved with Intel compiler is about 2 with 4 threads. The Sun Studio compiled version fails to scale at all on Harpertown.

In order to get a better understanding of this observation, we run a limited number of tests on a 4-socket Intel Dunnington[1] processor based machine with the same Sun Studio binaries to focus on binding and caching effects. In one test setting we bound all (up to four) threads to one socket using **taskset**, such that all threads shared the same L3 cache. In the other test setting we pinned each thread to a separate socket and therefore the threads had their own large L3 cache.

In the runs with shared cache we obtained a speedup of up to 1.8 which is similar to speedup of the Intel compiler version (see fig. 4 right). If the threads runs each on own socket we again don't see any speedup.

We therefore assume that the low scalability of the routine B is a consequence of memory bus limitations and cache issues. Unfortunately, we were not yet able to conduct any useful hardware performance counter experiments (see below).

[1] We choose a machine with 4 Intel Dunnington CPUs in comparison to a 2 socket Harpertown machine. Therefore it is possible to run up to 4 threads with a dedicated socket and thus without sharing cache.

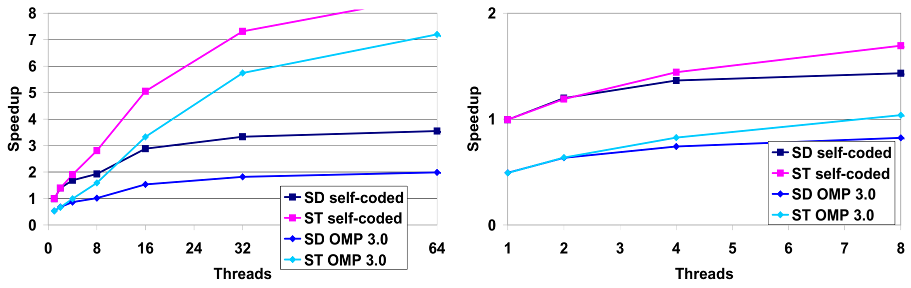


Fig. 5. Nested `PARALLEL SECTIONS` : Comparison of the self-coded and the OMP 3.0 driven implementations of thread forking boundaries on Routine B
Left: Niagara2, studio f90 express (**-fast**)
Right: Xeon, ifort 11.1.016 (**-O3**)

Evaluation of the new OpenMP 3.0 runtime functions to limit the maximum thread count in nested parallel `SECTIONS`

- The OpenMP 3.0 ICVs `THREAD-LIMIT-VAR` and `MAX-ACTIVE-LEVELS-VAR` are global the program, i.e. all simultaneous parallel regions share them. Furthermore there is no way to adjust the `THREAD-LIMIT-VAR` ICV from within the program (this ICV may be adjusted by setting the `OMP_THREAD_LIMIT` environment variable only) thus *all* parallel regions in the whole program share their value. Our own implementation is more flexible in adjustment of number of threads in parallel regions.
- The OpenMP 3.0 variants are potentially dangerous in use. Due to high default values of `THREAD-LIMIT-VAR` and `MAX-ACTIVE-LEVELS-VAR` for the tested compilers (e.g. 2147483647 for both ICVs by Intel compiler), simply neglecting to set an environment variable may lead to an heavily overloaded system or to running out of memory.

For routine A the performance of the hand-coded version and the OpenMP 3.0 runtime functions has been observed to be the same.

For routine B the OpenMP 3.0 implementation is approximately twice as slow as the hand-coded version (see fig. 5). Surprisingly, for this routine the tasking variant is also approx. two times slower with only one thread than the serial version. This behavior lets us draw the conclusion that there is an fundamental problem with this routine in combination with OpenMP 3.0. Here further research is needed.

The impact of limiting the number of tasks was also investigated.

We implemented reducing the task number by limiting the maximum task nesting depth both with `IF` clauses in task definition and with Fortran conditional branches and calling the serial version of the routine. In routine B, in which the most work is concentrated in the leaves, we implemented also the

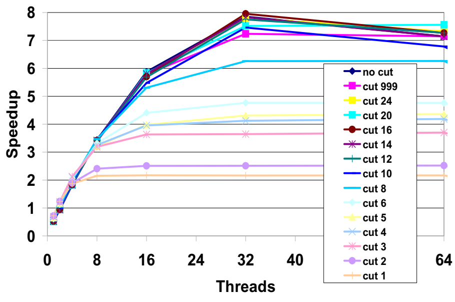


Fig. 6. Reducing the number of tasks by limiting the max. task nesting depth with Fortran conditional branching and calling the serial version of the routine. Niagara2, Routine B, Sun Studio f90 Express (**-fast**).

differentiation of inner nodes (for which no task were spawned) and leaves (which are outsourced to tasks).

We discovered that reducing the number of tasks does not result in any further speedup for KegelSpan. That is in contrast to the result reported in [7] when parallelizing the computation of Fibonacci numbers. We also see no overhead due to additional IF's if the cutting depth was chosen large enough to let the cutting condition always be false (and thus producing an task for every node in the tree). Only the slow-down due of advancing serialization and load imbalance was seen, if the cutting depth was decreasing, see fig. 6.

It seems that the task granularity in our program is considerably higher than the overhead of the runtime's task management.

In general, our performance measurements confirm that the tasking and adaptive approaches scale better than nested `PARALLEL SECTIONS` as the trees are unbalanced. Interestingly, in routine B the tasking approach is always much slower with few threads (a factor of two with a single thread) but it catches up when increasing the thread count. For the nested `PARALLEL SECTIONS` approach it is not clear whether limiting the nesting depth or the thread pool size is advantageous. For the adaptive parallelization the LIFO queue outperforms the FIFO queue for routine B, while for routine A the difference is negligible. The advantage of LIFO queue compared to FIFO seems to be evoked by better cache utilization. The last element put into the LIFO queue will probably immediately be processed, keeping the caches hot.

4.4 Pitfalls

By now, the support for OpenMP 3.0 is still in its infancy.

We had quite some difficulties to generate correct and efficient executables with the Intel 11.0 (beta and production) compilers when using tasks. We reported multiple compiler bugs but also found workarounds. Initially, we had to turn off optimization to generate correct executables. Since version 11.1.016 of the Intel beta compiler we are able to turn the optimization on by **-O3**, but we were forced to disable the production of debugging symbols by omitting the **-g** flag due of yet another bug in the compiler. Fixes for our problems are announced by Intel. The Sun Studio Express Compiler (Version 07/2008) came out a little later than first Intel compiler with tasking support and did not reveal any similar weaknesses when employing tasks.

By now, there are no tools available which may help to check the correctness of threaded programs which use tasks. We tried out Intel's Thread Checker and Sun's Thread Analyze without success, which is not surprising because these tools officially do not support tasking by now.

We also failed to get meaningful hardware counter data with Intel VTune due to many bugs and the inability of VTune to read the debug information from a binary compiled with the Sun Studio compiler. This prevented us from drilling further down into some of the performance peculiarities that we are not yet able to explain satisfactorily.

5 Conclusion

In the context of the simulation of bevel gear cutting processes we compared different approaches of parallelizing recursive algorithms for binary tree traversals using OpenMP with advanced Fortran 90/95.

Recursively nesting of `PARALLEL SECTIONS` is appealing because of it's simplicity, while the laborious stack implementation [5] performs better on unbalanced trees. The new tasking approach provided by OpenMP 3.0 combines ease of use of the first approach and the scalability of the second, provided that data scoping is handled with care.

We obtained a speed-up of up to 3.86 on the single-socket 8-core multithreaded Niagara 2-based system and up to 3.49 on the dual-socket 4-core Harpertown-based machine for the first routine ("A") and a speed-up of up to 8.59 on the Niagara 2- but only 1.98 on the Harpertown-based system for the second routine ("B").

Future work will include a test of upcoming compiler versions and tuning the runtime environment. Particularly, we want to investigate the matter of limited scalability of the routine B on Harpertown. Further experiments concerning untied tasks, thread placement, and wait policies will be carried out once the compilers mature and generate more reliable code.

References

1. WZL: Homepage of laboratory for machine tools and production engineering (wzl) of rheinisch-westfälischen technischen hochschule zu aachen, `http://www.wzl.rwth-aachen.de`
2. WZL: Calculation of the chip creation parameters for the manufacturing of bevel gears, `http://www.wzl.rwth-aachen.de/en/cd77f1f9bc96139dc125707e00445cb1.htm`

3. Brecher, C., Klocke, F., Schröder, T., Rütjes, U.: Analysis and simulation of different manufacturing processes for bevel gear cutting. In: Proceedings of the International Conference on Manufacturing, Machine Design and Tribology (ICMDT 2007), Printed by JSME, Sapporo, Japan (July 2007)
4. Klocke, F., Brecher, C., Rütjes, U.: Manufacturing simulation of the cutting process for bevel gears. In: Proceedings of the International Conference on Gears. VDI Berichte, vol. 1904, pp. 865–879. VDI Verlag GmbH, Duesseldorf (2005)
5. Kapinos, P.: Diploma thesis of Recursions in the multilevel parallelization of simulating machine tools and cutting processes with openmp (in german) (November 2007), http://www.sc.rwth-aachen.de/Diplom/PaulKapinos.html
6. OpenMP ARB: Latest official openmp specifications: Version 3.0 - (May 2008), http://www.openmp.org/mp-documents/spec30.pdf
7. Terboven, C.: First experiments with tasking in openmp 3.0 (June 2008), http://terboven.spaces.live.com/blog/cnsEA3D3C756483FECB316.entry

Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program Using OpenMP

Toshihiro Hanawa, Mitsuhisa Sato, Jinpil Lee, Takayuki Imada, Hideaki Kimura, and Taisuke Boku

University of Tsukuba, 1-1-1 Tennodai, Tsukuba 305-8577 Japan
{hanawa,msato,taisuke}@cs.tsukuba.ac.jp
{jinpil,imada,kimura}@hpcs.cs.tsukuba.ac.jp

Abstract. Recently, multicore technology has been introduced to embedded systems in order to improve performance and reduce power consumption. In the present study, three SMP multicore processors for embedded systems and a multicore processor for a desktop PC are evaluated by the parallel benchmark using OpenMP. The results indicate that, even if the memory performance is low, applications that are not memory-intensive exhibit large speedups by parallelization. The results also indicate a large performance improvement due to parallelization using OpenMP, despite its low cost.

1 Introduction

Recently, the use of embedded systems with complicated functions, such as digital home appliances and car navigation systems, has become widespread. These systems require increasingly higher performance as the functionality of the user interface becomes increasingly higher and the amount of information being processed increases. On the other hand, the power consumption of embedded systems must be reduced in order to extend the operating time of mobile devices and realize more environmentally friendly products. Therefore, multicore technology has been introduced not only for high-end general-purpose processors but also for embedded processors, because multicore is advantageous in terms of performance per watt. On a multicore SMP processor, conventional multi-thread programming using POSIX thread library has been required, or special assembly language code has been inserted at each synchronization point. However, as the applications on embedded system become larger, in order to reduce the development time an easier programming method for parallel processing is required in place of complex programming methods that require and skilled programmers.

OpenMP[1] is a portable programming model that provides a flexible interface for developing parallel applications on shared memory multiprocessors. By inserting a hint for parallelization, called a "directive", into a program written in a conventional programming language, such as C, C++, or Fortran, the OpenMP compiler generates the parallelized code. Recently, OpenMP has become available for several compiler products for desktop computers or servers.

In this way, the development environment of OpenMP can be used broadly. The GNU Compiler Collection (GCC), which is a well-known open-source compiler suite [2], has supported OpenMP since GCC version 4.2. Nevertheless, in the field of embedded systems, a special cross compiler is provided by the vendor, and most of these compilers cannot treat OpenMP directives.

In the present study, we evaluate some embedded multicore processors with a shared memory mechanism by parallel benchmark programs using OpenMP. In addition, we develop the OpenMP implementation using a cross compiling environment for embedded multicore systems. Since embedded multicore processors are widely used, we investigate the following:

– the effect of OpenMP as a programming method, and
– the memory bandwidth and synchronization performance as an SMP multicore processor and their impact on the overall system performance.

2 Multicore Processor for Embedded Systems

In the present study, we evaluate three multicore processors for embedded systems with shared memory: M32700, MPCore, and RP1. Furthermore, as an example of a multicore processor for a desktop PC, the Intel Core2Quad Q6600 is examined for the purpose of comparison.

In the following, we will describe the M32700, the MPCore, and the RP1 in greater detail.

2.1 Renesas Technology M32700

The M32700[3], developed by Renesas Technology, is a dual-core processor embedded with two M32R-II cores.

Figure 1 shows a block diagram of the M32700 processor. The M32R-II core consists of seven-stage pipeline, and each core includes an 8-Kbyte instruction

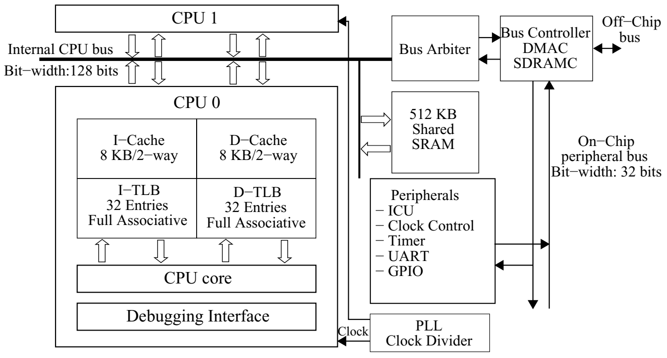


Fig. 1. Block diagram of the M32700 processor[3]

cache and an 8-Kbyte data cache with two-way set associativity. The data cache is managed by a cache consistency mechanism between the cores. On the M32700 chip, a 512-Kbyte SRAM is shared with two cores via a 128-bit internal bus.

The M32R-II instruction set consists of 32-bit instructions and more frequently used 16-bit instructions. If two 16-bit instructions are located continuously, these instructions can be issued simultaneously depending on the combination of instructions. On the other hand, the M32700 does not have a floating-point function unit.

2.2 ARM/NEC Electronics MPCore

The MPCore[4], developed by ARM and NEC Electronics, is a quad-core processor into which four MP11 cores are embedded. The MPCore is designed as a combination of various function modules called PrimeCells to provide a flexible configuration. Figure 2 shows a block diagram of the MPCore processor.

The MP11 core is an implementation of the ARM11 micro architecture based on ARMv6 instruction set architecture and has an eight-stage, single-issue pipeline. The MP11 core consists of the 32-bit ARM instruction set, the 16-bit Thumb instruction set, the Jazelle instruction set for acceleration of the Java virtual machine, as well as DSP extensions, and SIMD instructions, for example. In the present study, we use the only the 32-bit ARM instruction set.

Each core is connected to a Snoop Control Unit (SCU), and two AXI buses from the SCU are connected to external devices. In the evaluation board used in

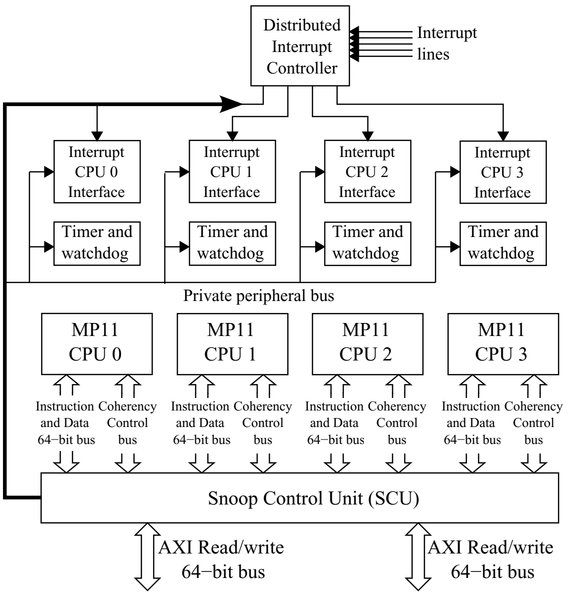

Distributed Interrupt Controller
Interrupt lines
Interrupt CPU 0 Interface
Interrupt CPU 1 Interface
Interrupt CPU 2 Interface
Interrupt CPU 3 Interface
Timer and watchdog
Timer and watchdog
Timer and watchdog
Timer and watchdog
Private peripheral bus
MP11 CPU 0
MP11 CPU 1
MP11 CPU 2
MP11 CPU 3
Instruction and Data 64−bit bus
Coherency Control bus
Snoop Control Unit (SCU)
AXI Read/write 64−bit bus
AXI Read/write 64−bit bus

Fig. 2. Block Diagram of the MPCore processor[4]

the present study, external peripheral devices, including two AXI bus interfaces for attachment to the MPCore processor and a DRAM controller, are incorporated into a single FPGA chip. Therefore, the external system bus and the DRAM controllers are operated at a clock frequency of 30 MHz. To reduce the influence of the performance degradation due to the slow frequency for the bus clock, in addition to 32-Kbyte instruction and 32-Kbyte data level-1 caches, a level-2 shared-cache of 1 Mbyte is embedded on the MPCore[5].

2.3 Renesas/Hitachi/Waseda University RP1

The RP1 processor, developed by Renesas, Hitachi, and Waseda University, is a quad-core processor based on SH-X3 architecture[6]. The RP1 includes four SH-4A cores and performs 4,320 MIPS and 16.8 GFlops at a clock frequency of 600 MHz. Figure 3 shows a block diagram of the RP1 processor.

The SH-4A core consists of an eight-stage pipeline and can issue two 16-bit instructions simultaneously. The RP1 has a dedicated snoop bus for cache consistency, and data transfer for cache coherence control can avoid traffic on the Super Highway (SHwy). The RP1 also includes local memories dedicated to each core that can be accessed with one clock, ILRAM for instructions and OLRAM for data, a local memory that can be accessed with several clocks, URAM, and a centralized shared memory, CSM.

In the present study, we use Linux as the operating system and do not apply the internal memory in the RP1 chip to Linux, so that the RP1 is treated as a conventional SMP processor.

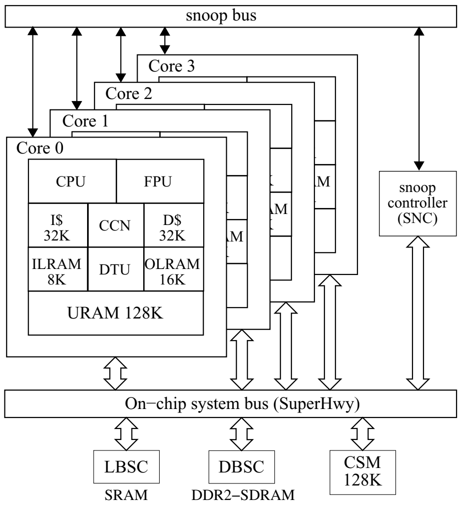


Fig. 3. Block diagram of the RP1 processor[6]

3 Omni OpenMP Compiler

As an OpenMP implementation, Omni OpenMP compiler version 2, which we developed previously, is used. In order to adapt to each multicore architecture, the runtime library must be implemented using spinlock, and the cross compiling environment must be built as described below.

3.1 Implementation of the Runtime Library

When an OpenMP program is compiled using Omni OpenMP compiler, we can choose either `mutex_lock`, provided by POSIX thread, or a dedicated spinlock function for specific architecture from the runtime libraries as the mutual exclusion. For this purpose, we implement the runtime libraries using spinlock for each of the multicore processors examined in the present study. Initially, a lock variable is set to 0 and is changed to 1 when the lock is acquired. If the lock variable is already 1, the lock is in use, and the process waits using busy waiting until the release of the lock. At the exit of the critical section, the lock variable reverts to 0, i.e., the unlocked state. In addition, to avoid false sharing of the cache block, lock variables are aligned to the boundary of cache line.

Figure 4 shows the implementation of the spinlock function for each multicore in assembly language. These implementations are performed by referencing the Linux kernel[7].

M32700 M32R provides only bus lock/unlock operations, `lock`/`unlock` instructions, as the atomic operation. While the bus is locked, the other processor or peripherals cannot access the bus, which causes degradation of performance. Here, we assume that the address of the lock variable is stored in `r0`.

To minimize the lock period, after the `lock` instruction reads the lock variable to `r5`, the `unlock` instruction is immediately performed, and the lock variable is set to 1 simultaneously. If `r5` is not 0, then the lock is already in use, and `r5` returns to the starting point. To avoid the bus contention caused by repeating the lock operation, the lock variable is tested before the second lock operation, and the `nop` (no-operation) instruction is added.

M32700	MPCore	RP1
`ldi r4,#1`	`mov r3, #1`	`loop:`
`loop:`	`loop:`	`movli.l @r4, r0`
`lock r5,@r0`	`ldrex r1, [r0]`	`tst r0, r0`
`unlock r4,@r0`	`teq r1, #0`	`bf loop`
`bnez r5,loop`	`strexeq r1, r3, [r0]`	`mov #1, r0`
`exit:`	`teqeq r1, #0`	`movco.l r0, @r4`
	`bne loop`	`bf loop`
	`exit:`	`exit:`

Fig. 4. Spinlock codes for each architecture

MPCore ARMv6 architecture provides the Exclusive Load, `ldrex` instruction, and the Exclusive Store `strex` instruction. Here, we assume that the address of the lock variable is stored in `r0`.

`Ldrex` reads the lock variable to `r3`, and then the operation mode is changed to the exclusive access mode. In line 4, if `r1` is 0, then the Z (Zero) flag is set. Hereinafter, suffix `eq` indicates that the instruction is executed if and only if the Z flag is set. Otherwise, the instruction is replaced with nop. In line 5, when the Z flag is set, that is, when the lock acquisition is successful, `strex` tries to write the value 1 into the lock variable. If another processor has not written to the lock variable until the exclusive access mode, then the write operation by `strex` is successful, and `r1` is set to 0. Otherwise, `r1` becomes 1. In line 6, `teqeq` checks whether store is successful. If the lock cannot be acquired, or the store is unsuccessful, the operation returns from line 7 to line 3.

RP1 On SH4A architecture, Move Linked, `movli`, and Move Conditional, `movco` instructions are available. The `movli` and `movco` operations are similar to Exclusive Load and Exclusive Store, respectively, of the MPCore. Here, we assume that the address of the lock variable is stored in `r4`.

`Movli` reads the lock variable to `r0` implicitly. Unless the value `r0` is 0, the lock acquisition is retried in lines 3 and 4. `Movco` tries to set the lock variable to 1. The T (True) flag is set if this is successful. Otherwise, the T flag is reset, and the operation returns from line 7 to line 2 `movli`.

3.2 Cross Compiling Environment

The Omni OpenMP compiler consists of a front-end part, a translation part, and a runtime library. The front-end part converts a C or Fortran program to a common intermediate language. The translation part translates the OpenMP directives into a normal multithreaded C program using POSIX threads.

In the case of cross compiling, similar to the native compiler, the front-end part and the translation part can be executed on the host PC (x86) because these processes and codes are architecture-independent. Finally, the cross compiler for the target architecture generates the target binary linked with the runtime library described in Section 3.1 from the multithreaded code.

4 Performance Evaluation

4.1 Evaluation Environment

In the present study, we evaluate four multicore processors described in Section 2. Table 1 shows the specifications and the operating environment. We adopt Linux 2.6 as the operating system. These embedded systems start up via 100-Mbps Ethernet using the Network File System (NFS), and the Q6600 in the desktop PC uses the local file system.

As the OpenMP implementation, Omni OpenMP compiler Version 2 is commonly used for all platforms in the present study. For the back-end compiler, we

Table 1. Evaluation environment

Processor	M32700	MPCore	RP1	Q6600
# of Cores	2	4	4	4
Frequency Core/Int./Ext. Bus	300/75/75 MHz	210/210/30 MHz	600/300/50 MHz	2.4 GHz
Cache L1: I+D	8K+8K/2-way	32K+32K/4-way	32K+32K/4-way	32K+32K/8-way
L2:	—	1M/8-way	—	4M×2/16-way
Line size (byte)	16	32	32	64
Memory	32-Mbyte	256-Mbyte	128-Mbyte	2-Gbyte
Type	SDRAM	DDR	DDR2-600	DDR2-800
Clock	100 MHz	30 MHz	300 MHz	400 MHz
OS	Linux 2.6.25	Linux 2.6.19	Linux 2.6.16	Linux 2.6.18
`uname -m`	m32r	armv61	sh4a	i686
C Compiler	gcc 4.1.2 20061115	gcc 4.1.1	gcc 3.4.5 20060103	gcc 4.1.2 20061115
Target	`-m32r2`	`-mcpu=mpcore`	`-m4a`	`-march=nocona`
C Library	glibc 2.3.6.ds1-13	glibc 2.3.6	glibc 2.3.3	glibc 2.3.6.ds1-13.etch5
Pthread	Linuxthreads 0.10	NPTL 2.3.6	Linuxthreads 0.10	NPTL 2.3.6

use various versions of gcc C compiler, as listed in Table 1. We also use glibc as the standard C library. However, each environment provides a POSIX thread library through different implementations.

LinuxThreads is the first implementation of POSIX thread on Linux. In order to manage thread creation or thread termination, a manager thread is required independently of the computation thread, and operations related to synchronization are implemented by signal. In contrast, the Native POSIX Threads Library (NPTL) is an implementation that was developed to solve the problem of LinuxThreads. NPTL uses futex, which stands for fast user-level locking mechanism. In the case of the RP1, although the TAS (test-and-set) instruction is not suitable for the SMP environment, the LinuxThreads library contains TAS (test-and-set) instructions. Therefore, we rewrite the library using MOVLI/MOVCO instructions and replace the pthread library of the system with the improved library described herein.

4.2 Benchmark

Next, we explain the parallel benchmarks used in the present study.

The MiBench suite[8] is a free benchmark suite for embedded processors that emulates the benchmark suite developed by the EEMBC (Embedded Microprocessor Benchmark Consortium)[9]. MiBench includes six categories, namely, Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications, and consists of 35 benchmarks. In the present study, we consider the major algorithms to parallelize benchmarks using OpenMP for typical applications on embedded systems. We choose Susan Smoothing (SS), the image-processing algorithm from Automotive@category, Blowfish Encrypt (BF) from Security category, and FFT from Telecommunications category, and parallelize these benchmarks using OpenMP.

In addition, we attempt to confirm the performance of target multicore processors using existing scientific applications. We use some benchmarks from the NAS Parallel Benchmark (NPB)[10], including IS and CG. IS in NPB3.3-OMP[11] is written in C language with OpenMP. Although CG in NPB is described in Fortran, Omni OpenMP provides a modified version of CG, which is translated into C language and uses OpenMP directives as the sample.

MediaBench is a benchmark for embedded multimedia applications [12]. We use the parallelized version using OpenMP of mpeg2encode [13]. Since we consider multimedia streaming to be one of the most suitable applications for embedded multicore processors, we choose this benchmark.

4.3 Performance Evaluation for Synchronization

First, we preliminarily evaluate each system for synchronization using syncbench of the EPCC micro benchmark[14]. Table 2 shows the results for syncbench. The M32700 is investigated using two cores, and the other processors are investigated using up to four cores. As a result, the M32700 and the RP1 using LinuxThreads as the POSIX thread library have very low performance in the cases of "single," "critical," "lock/unlock," and "atomic" with mutex_lock, which indicates that the signal handling degrades the performance when rendezvous occurs among the threads. To solve this problem, we introduce the spinlock mechanism for synchronization (in place of mutex_lock). The results indicate that the introduction of the spinlock in LinuxThreads results in an extreme speedup. For

Table 2. Results of EPCC syncbench (Unit: μs)

	M32700 (2 PU)		MPCore (4 PU)		RP1 (4 PU)		Core2Quad (4 PU)	
	Mutex	Spin	Mutex	Spin	Mutex	Spin	Mutex	Spin
parallel	392.2	376.8	436.5	434.8	107.8	107.2	2.80	2.25
for	18.5	13.6	7.46	6.15	1.66	1.42	0.364	0.372
parallel for	399.7	383.6	436.3	435.7	108.2	107.7	3.71	2.47
barrier	14.1	10.7	6.11	5.98	1.13	0.867	0.301	0.316
single	50.8	9.87	3.14	3.12	295.1	1.53	4.54	0.859
critical	273.5	3.15	0.921	0.837	128.2	0.190	1.31	0.129
lock/unlock	273.1	2.51	1.03	0.962	121.0	0.174	1.41	0.131
ordered	8.64	7.08	1.50	1.33	0.584	0.598	0.191	0.168
atomic	241.0	0.501	0.894	0.893	121.0	0.365	0.474	0.307
reduction	401.9	387.1	443.9	443.8	327.0	109.1	6.13	3.35

example, the speedup by spinlock is 482 in the case of “atomic” on the M32700, and 695 in the case of “lock/unlock” on the RP1. On the MPCore using NPTL, in the case of “ordered”, the performance of spinlock increases slightly (1.12 times faster) compared with that of mutex_lock. Based on these results, hereinafter, we use only spinlock for synchronization. In the case of embedded processors, “parallel,” “parallel for,” and “reduction” directives, which include the element of “parallel” directive, the execution time is relatively large. When the “parallel” directive is assigned, some threads are generated so that the program can be performed in parallel, and memory access is frequent. The MPCore, for which the access time to DRAM is slowest, requires the longest execution time for parallel directives. Naturally, the parallel region should be assigned to the largest section possible in order to reduce the overhead for thread assignment.

4.4 Evaluation by Parallel Benchmarks Using OpenMP

Next, we evaluate the multicore processors using parallel benchmarks. First, we use NAS Parallel Benchmarks. Figure 5 shows the speedup of IS (Class W). In addition, Table 3 shows the actual execution times for all of the benchmarks. NPB IS is a memory-intensive program. Since the embedded processors have low bandwidth for memory access, the MPCore with four cores only achieves a speedup of 1.6 times, whereas the speedup of the M32700 is 1.1 times. In contrast, the speedup of the RP1 with four cores is 2.9 times. The RP1 has a dedicated bus for the snoop cache control, and the above results indicate that the mechanism for cache control in the RP1 is effective. Figure 6 shows the speedup of CG ($1,400 \times 1,400$,sparse matrix). On this benchmark, the speedups of both the RP1 and the Q6600 with four cores are 2.8 times. The reason for this is that CG is a computation-intensive benchmark. On the other hand, in the case of

Table 3. Results of Performance Evaluation (Unit: seconds)

Processor	PU	IS	CG	Susan_s	FFT	Blowfish	Mpeg2encode
M32700	1	4.66	183.5	22.1	19.6	175.3	1143.8
	2	4.32	93.2	11.8	10.5	94.3 (90.6–102.4)	788.0
MPCore	1	11.3	31.0	21.7	3.19	138.9	—
	2	7.66	15.6	11.2	1.71	112.4 (69.8–140.6)	—
	3	7.09	10.8	7.80	1.31	79.4 (46.8–139.8)	—
	4	7.09	8.15	5.99	1.03	76.1 (34.9–139.3)	—
RP1	1	4.14	4.43	8.49	0.280	60.2	57.4
	2	2.26	2.44	4.51	0.147	33.5 (29.5–60.0)	46.6
	3	1.64	1.85	3.18	0.107	31.6 (19.9–59.8)	39.8
	4	1.41	1.56	2.52	0.085	23.5 (19.7–30.6)	35.9
Q6600	1	0.06	0.215	0.933	0.0076	6.12	5.47
	2	0.036	0.116	0.476	0.0048	3.14 (3.13–3.15)	3.69
	3	0.029	0.086	0.322	0.0065	2.59 (2.58–2.62)	2.88
	4	0.026	0.073	0.251	0.0063	1.70 (1.69–1.71)	2.48

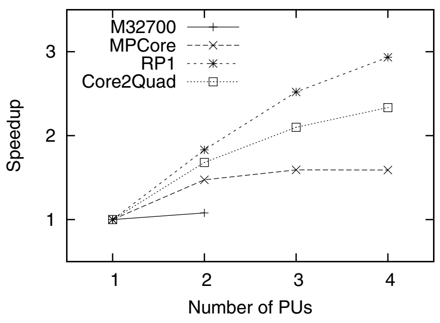


Fig. 5. Speedup of NPB IS (CLASS=W)

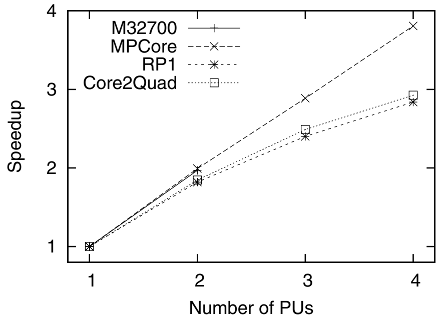


Fig. 6. Speedup of CG

the MPCore, the speedup is 3.8 times. The cache efficiency appears to increase because the L2 cache is shared among four cores.

Next, we investigate the speedup in the case of MiBench. Here, we introduce MiDataSets, which is a collection of various workloads for MiBench[15]. We adopt `19.pgm` for SS, `4.txt` for BF from MiDataSets, and the large dataset in MiBench is used for FFT (nwave=6, nsample=65,536). In addition, we change the algorithm in BF from the CFB64 mode to the ECB mode. Figures 7, 8, and 9 show the speedups of SS, BF, and FFT, respectively. The speedup of SS is from 3.4 to 3.7 times due to high parallelism. The parallel version of BF performs the following operations in a pipelined manner: read 40 bytes from input file, encrypt 100 times, then write 40 bytes to output file. Since BF indicates scattering results of the execution time (except for the Q6600), we append the minimum and maximum execution times to the average time. This dispersion appears to be caused by the influence of the NFS because the results for the Q6600 are stable. As a result of the slight dependency among only 8 bytes in the ECB mode, in the best case of all the trial, we can obtain a speedup of 4.0 times with four cores on the MPCore, and a speedup of 3.0 times with three cores on the RP1.

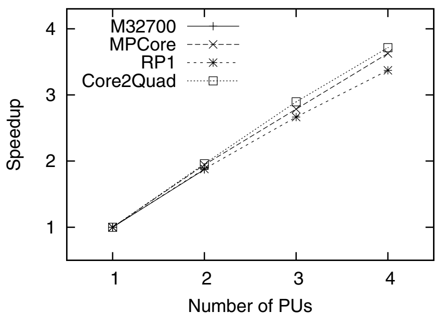


Fig. 7. Speedup of Susan (smoothing)

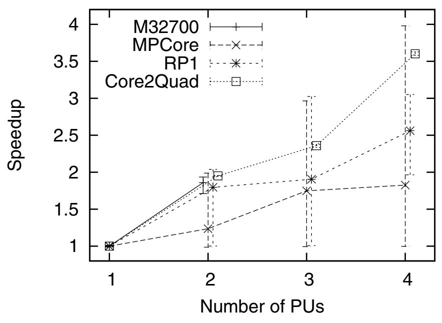


Fig. 8. Speedup of Blowfish

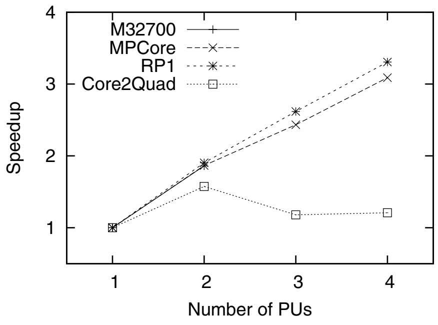


Fig. 9. Speedup of FFT

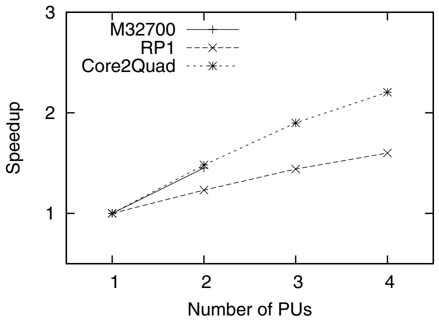


Fig. 10. Speedup of Mpeg2encode

For FFT on the RP1 with four cores, a speedup of 3.3 times is obtained due to high parallelism. For the Q6600, the speedup is degraded in the case of three or four cores, as compared to two cores, because the working set is so small for the Q6600 that the synchronization cost becomes dominant. Table 4 shows the cost of modification for OpenMP. Parallelization through the use of OpenMP is quite simple, and involves only the insertion of a small number of OpenMP directives.

Finally, Figure 10 shows the speedup of mpeg encoding. We cannot execute this benchmark on the MPCore because of a fatal error. Although the speedup on the Q6600 with four cores is 2.2 times, in the case of the RP1 with four cores, the speedup is only 1.6 times due to the overhead of file operation via NFS. In the case of the M32700, there is no degradation of the performance because the execution time is sufficiently long due to the software emulation of floating point operations.

Table 4. Modification Cost for OpenMP

Benchmark	Change
SS	add 6 directives
BF	add 9 directives & modify 12 lines
FFT	add 4 directives

5 Related Work

Several embedded multicore processors had been evaluated indivisually. Hotta, et al. [13] evaluated the M32700 by mpeg2enc in MediaBench parallelized with OpenMP. Blume, et al. [16] evaluated the MPCore using several applications parallelized with OpenMP. Seo, et al. [17] studied OpenMP directive extension for BlackFin 561 Dual-Core processor, and they modified EEMBC benchmarks[9] for parallel version using OpenMP.

Miyamoto, et al. [18] evaluated Fujitsu FR1000 processor and the RP1 using multimedia applications parallelized automatically by OSCAR compiler which they developed.

Several different kinds of embedded multicore processors had never been evaluated simultaneously by the parallel benchmark using OpenMP.

6 Conclusion

In the present study, we evaluated the performance of four multicore processors, namely, the M32700, the MPCore, and the RP1 for embedded systems, and the Core2Quad Q6600 for a desktop PC. After we investigated the synchronization performance using syncbench of the EPCC micro benchmark, we used various benchmarks, including the NAS Parallel Benchmarks, MediaBench, and MiBench, which are parallelized by OpenMP.

As a result, although embedded multicore processors have larger synchronization cost and slower memory performance than multicore processors for desktop PC, the spinlock mechanism enables embedded multicore processors to improve the synchronization performance. Moreover, according to the parallel benchmarks using OpenMP, the performance became higher as the number of cores increased for the case in which several OpenMP directives were inserted into the source code. Therefore, OpenMP is very useful for parallelizing embedded applications as well as scientific applications on HPC. On the other hand, major hurdles remain with respect to the use of OpenMP for embedded systems. For example, parallel processing using OpenMP is too difficult to satisfy real-time restriction. To apply OpenMP to embedded systems, some extensions for OpenMP directives will be required, including functions such as real-time processing and power-awareness.

In the future, the effect of spinlock for synchronization under multiple parallel-workloads should be examined. In addition, most multicore processors include fast internal memories on the chip, and we will consider using these internal memories to speedup synchronization. Furthermore, we will investigate the transition of power consumption of each multicore processor system for various numbers of cores.

Acknowledgment

The authors would like to thank Renesas Technology Corporation for providing the M3T-M32700UT. The RP1 multicore-chip and its runtime environment used in the present study were developed by Waseda University (Kasahara-Kimura Laboratory), Renesas Technology Corporation, and Hitachi, Ltd. as part of the NEDO project entitled “Multi-core Processor Technologies for Real Time Consumer Electronics.” The present study was supported in part by the JST/CREST program entitled “Computation Platform for Power-aware and Reliable Embedded Parallel Processing System” in the research area of “Dependable Embedded Operating Systems for Practical Use.”

References

1. The OpenMP Architecture Review Board: The OpenMP API specification for parallel programming,
`http://openmp.org/wp/`
2. Free Software Foundation: GCC, the GNU Compiler Collection,
`http://gcc.gnu.org/`
3. Kaneko, S., et al.: A 600MHz Single-Chip Multiprocessor with 4.8GB/s Internal Shared Pipelined Bus and 512kB Internal Memory. In: International Solid-State Circuits Conference (ISSCC), vol. 1, pp. 254–255 (2003)
4. ARM Limited: ARM11 MPCore Processor Technical Reference Manual (2006)
5. ARM Limited: Using a CT11MPCore with the RealView Emulation Baseboard (2006)
6. Yoshida, Y., et al.: A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption. In: International Solid-State Circuits Conference (ISSCC), pp. 100–590 (2007)
7. The Linux Kernel Archives, `http://kernel.org/`
8. Guthaus, M., et al.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization (2001)
9. The Embedded Microprocessor Benchmark Consortium: EEMBC — The Embedded Microprocessor Benchmark Consortium, `http://www.eembc.org/`
10. Bailey, D., et al.: The NAS Parallel Benchmarks. RNR Technical Report RNR-94-007, NASA Ames Research Center (1994)
11. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NAS System Division NASA Ames Research Center (1999)
12. Lee, C., Potkonjak, M., Mangione-Smith, H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: International Symposium on Microarchitecture (Micro-30), pp. 330–335 (1997)
13. Hotta, Y., Sato, M., Nakajima, Y., Ojima, Y.: OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor. In: Sixth European Workshop on OpenMP (EWOMP 2004), pp. 37–42 (2004)
14. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: European Workshop on OpenMP (EWOMP 1999), pp. 99–105 (1999)
15. Fursin, G., Cavazos, J., O'Boyle, M., Temam, O.: MiDataSets: Creating The Conditions For A More Realistic Evaluation of Iterative Optimization. In: De Bosschere, K., Kaeli, D., Stenström, P., Whalley, D., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4367, pp. 245–260. Springer, Heidelberg (2007)
16. Blume, H., von Livonius, J., Rotenberg, L., Noll, T.G., Bothe, H., Brakensiek, J.: OpenMP-based parallelization on an MPCore multiprocessor platform - A performance and power analysis. Journal of Systems Architecture 54(11), 1019–1029 (2008)
17. Seo, H., Kim, S.W.: OpenMP Directive Extension for BlackFin 561 Dual Core Processor. In: Sixth IEEE International Conference on Computer and Information Technology (CIT 2006), p. 49 (2006)
18. Miyamoto, T., Asaka, S., Mikami, H., Mase, M., Wada, Y., Nakano, H., Kimura, K., Kasahara, H.: Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API. In: International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008), pp. 600–607 (2008)

Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore*

Chunhua Liao[1], Daniel J. Quinlan[1], Jeremiah J. Willcock[2], and Thomas Panas[1]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,quinlan1,panas2}@llnl.gov
[2] Computer Science Department
Indiana University
Lindley Hall Room 215
150 S. Woodlawn Ave.
Bloomington, IN, 47404
jewillco@osl.iu.edu

Abstract. Automatic introduction of OpenMP for sequential applications has attracted significant attention recently because of the proliferation of multicore processors and the simplicity of using OpenMP to express parallelism for shared-memory systems. However, most previous research has only focused on C and Fortran applications operating on primitive data types. C++ applications using high-level abstractions, such as STL containers and complex user-defined types, are largely ignored due to the lack of research compilers that are readily able to recognize high-level object-oriented abstractions and leverage their associated semantics. In this paper, we automatically parallelize C++ applications using ROSE, a multiple-language source-to-source compiler infrastructure which preserves the high-level abstractions and allows us to unambiguously leverage their known semantics. Several representative parallelization candidate kernels are used to explore semantic-aware parallelization strategies for high-level abstractions, combined with extended compiler analyses. Those kernels include an array-based computation loop, a loop with task-level parallelism, and a domain-specific tree traversal. Our work extends the applicability of automatic parallelization to modern applications using high-level abstractions and exposes more opportunities to take advantage of multicore processors.

1 Introduction

Today's multicore processors have been forcing application developers to parallelize legacy sequential codes and/or write new parallel applications if they

* This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We thank Dr. Qing Yi for her dependence analysis implementation in ROSE.

want to take advantage of shared-memory parallelism supported by hardware. However, parallel programming is never an easy task for users, given the stunning work to deal with extra issues in parallel computing, such as dependencies, synchronization, load balancing, and race conditions. Therefore, parallelizing compilers and tools are playing increasingly important roles in allowing the full utilization of new computer systems and enhancing the productivity of users.

OpenMP [1] is a simple and portable parallel programming model that extends existing programming languages like C/C++ and Fortran 77/90 to include additional parallel semantics. The extensions OpenMP provides contain compiler directives, user level runtime routines and environment variables. Programmers can use OpenMP to express parallelization opportunities and strategies for applications. Moreover, the simple API provided by OpenMP has attracted parallelizing compilers and tools to use OpenMP as a target for interactive or automatic parallelization.

Although numerous parallelizing compilers [2,3] and tools [4,5] have been presented during the past decades, most of them focus only on C and/or Fortran applications operating on primitive data types. On the other hand, object-oriented languages, especially C++, are widely used to develop scientific computing applications. Those applications are often written with various standard and/or user-defined high-level abstractions, such as those in the C++ Standard Template Library (STL), now part of the C++ standard. While high-level abstractions successfully hide their implementation details and are useful to users for this purpose, they significantly impede static code analyses applied to their complex implementation. Typically, significant information about the abstractions is lost during the compiler's lowering to a simple intermediate representation (IR). Thus, compilers are often forced to make conservative assumptions for applications using such abstractions and are not able to apply many optimizations, including automatic parallelization.

In this paper, we use a source-to-source compiler infrastructure, ROSE [6], to explore compiler techniques to recognize high-level abstractions and to exploit their semantics for automatic parallelization. Our goal is to automate the process of migrating existing sequential C++ applications to multicore machines and to assist in developing new parallel applications. Specifically, our work addresses the concerns of parallelism for three target audiences: 1) users with legacy code (C/C++) using standard abstractions (STL, etc.), 2) users and library writers with domain-specific abstractions that have semantic properties that match those of the ones we make available, 3) library developers who are developing domain-specific abstractions for users and leveraging the semantics using their own semantic specifications (ones that we don't define). Our work addresses the essential requirement that modern compilers be fundamentally extensible in a way that simplifies how domain-specific abstractions can be optimized.

The remainder of this paper is organized as follows. The ROSE compiler infrastructure is introduced in the next section. Section 3 discusses high-level abstractions and parallelization. Section 4 then presents the details of a semantic-aware parallelizer using ROSE. Preliminary results of our work are given in Section 5.

Section 6 discusses related work. Finally, Section 7 presents our conclusions and the future directions of this work.

2 The ROSE Compiler Infrastructure

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++ and Fortran applications. Since it preserves the representation of high-level abstractions, no required information to recognize such abstractions is lost and the associated semantics can be reliably inferred. ROSE allows even non-expert users to exploit compiler techniques to address the analysis and transformation of abstractions.

Fig. 1 illustrates a typical source-to-source translator built using ROSE. The Edison Design Group (EDG) front-end [7] is used to parse C and C++ applications. EDG source files and its IR are protected under commercial or research licenses, but may be distributed freely in binary form. Language support for Fortran 2003 (and earlier versions) is based on the open source Open Fortran Parser (OFP) [8] developed at Los Alamos National Laboratory. Using both EDG and OFP, ROSE presents a common object-oriented, open-source IR for C/C++ and Fortran. The ROSE IR includes an abstract syntax tree (AST), symbol tables, a control flow graph, etc. and is based loosely on the Sage++ IR design [9]. Also, a set of distributed symbol tables is associated with the AST tree to store symbols' information within each scope. Generic and custom program analysis and transformation can be built on top of the ROSE IR. The ROSE unparser generates source code in the original source language from the transformed AST, with all original comments and C preprocessor control structures preserved. Finally, a vendor compiler is optionally called to continue the compilation of the generated (transformed) source code, generating a final executable.

The ROSE AST, together with its corresponding symbol tables, fully supports type resolution, semantic analysis, and overloaded function resolution. All

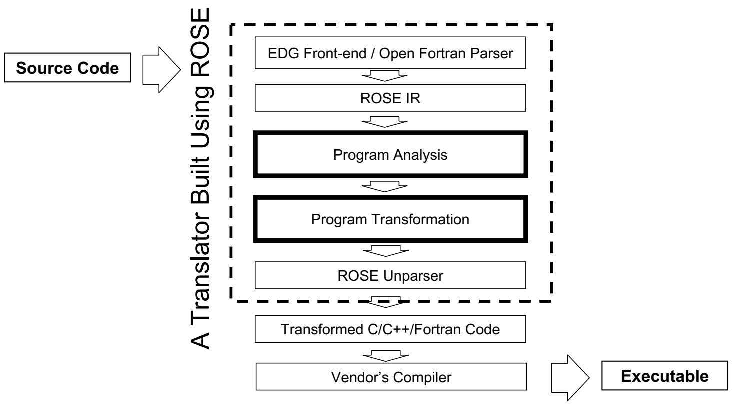


Fig. 1. A source-to-source translator built using ROSE

information in the application source code is preserved in the AST, including C preprocessor control structure, source comments, source position information, token stream (including whitespace), and C++ template information. The ROSE AST also has a rich set of interfaces for building source-to-source translators. These interfaces support efficient AST traversals, AST node queries, AST construction, copying, insertion, removal, and symbol table lookups. Moreover, persistent attributes are introduced in the AST to easily store and evaluate arbitrary user-defined information, including AST annotations. These attributes are persistent in that they are preserved when the AST is written out to (and read in from) a binary file.

A number of program analyses and transformations have been developed for ROSE. They are designed to be utilized by users via calling simple function interfaces. The program analyses available include call graph analysis, control flow analysis, data flow analysis (def-use chain, reaching definition, live variables, alias analysis etc.), class hierarchy analysis and dependence analysis. Representative program translations developed with ROSE are partial redundancy elimination, constant folding, inlining, outlining (separating out a portion of code as a function), and loop transformations (a loop optimizer supporting aggressive loop optimizations such as fusion, fission, interchange, unrolling and blocking).

3 High-Level Abstractions and Parallelization

General purpose languages typically permit the construction of abstractions; represented by functions, data structures, etc. These permit high-level representations of typically user-defined concepts. C++, as an object-oriented language, supports more complex abstractions and encourages the use of classes, member functions, templates, etc.

Knowledge of the semantics of the abstractions can be a short-cut for program analysis based on the implementation of an abstraction. In the case of complex abstractions with semantics hidden behind the use of pointers, leveraging known or published semantics of the abstractions can often be more productive. As an example, the knowledge that STL vectors are contiguous in memory is critical to numerous optimization opportunities, but it might be impossible to obtain from an analysis of a specific STL implementation because of the complexity of its internal pointer handling. By exploiting well-defined semantics of high-level abstractions, compilers can significantly enhance the applicability and accuracy of existing analyses and optimizations. Such work also serves to encourage libraries to define abstractions with well-defined semantics. For instance, traditional parallelization algorithms designed for primitive data types can be extended to handle applications using high-level abstractions if the applications demonstrate similar semantic properties. The semantics of abstractions often directly indicate the side effects of function calls and such knowledge can significantly benefit parallelization which is often disabled because the inability to accurately summarize read and write accesses hidden behind call sites.

In the following subsections, we examine several typical candidates and explore parallelization strategies for applications using high-level abstractions.

3.1 An Array-Based Computation Loop

Loops operating on fixed-sized arrays are probably the most popular and representative examples for automatic parallelization using OpenMP. Typically, an array-based computation loop parallelizable by using **omp parallel for** has the following properties:

1. The loop has a canonical form (**for** (init; test; incr) block) which satisfies the requirements as defined by the OpenMP specification.
2. The loop operates on arrays using contiguous memory locations for a set of elements of the same type.
3. The elements of arrays do not overlap in memory or alias each other.
4. Random element accesses with a constant cost can be achieved by calculating offsets from an array base using subscripts.
5. The operations on the arrays do not rearrange the memory layout of elements and invalidate their accesses using subscripts across different iterations.
6. There are no loop carried data dependencies for array element accesses.

Conventional parallelization algorithms rely on a set of transformations and analyses in order to judge the safety of parallelization. For example, loop normalization is conducted to produce a canonical form, if possible. Alias analysis is used to tell if there are aliased elements. A set of data dependence tests based on array subscripts are used to determine if different loop iterations are independent. Automatic parallelization can be extended to handle high-level abstractions by leveraging their semantics and applying the conventional analyses and transformations. We take the following STL vector computation loop as an example to explore a viable parallelization method. The method is generic so that it can be applied to other high-level abstractions with similar semantics, including the STL deque or user-defined types.

```
std::vector <int> v1(100);
for (int i = 0; i < 100; i++)
  v1[i] = v1[i] + i;
```

The STL vector has many semantics (e.g., iterator invalidation rules) which can be taken advantage of by automatic parallelization. As a sequential container with contiguous storage for its elements, it supports random element access via both iterators and member functions (**operator**[] and at()). Although a vector can be reallocated or resized during its lifetime, it is quite common to have computation phases in which the vector participates in computations as if it was a fixed-sized primitive array. Within these phases, the arguments of random element access functions can be directly treated as array subscripts and passed to relevant parallelization analysis, especially array dependence analysis. The elements of the vector have to be verified to be alias-free and non-overlapping, either by compiler analyses or user annotations. Even for a loop using random access iterators, an extended loop normalization phase can convert the loop into a canonical form that is friendly to parallelization. For example, `for(vector<T>::iterator i = v.begin(); i != v.end(); i++)` can be transformed to `size_t n = v.size(); for (size_t i = 0; i < n; i++)`. Dereferences of the iterator within the loop body can be replaced with equivalent element access function calls. In this case, all variable accesses like `(*i)` and `i[n]` are replaced with `v[i]`(or `v.at(i)`) and `v[i + n]` (or `v.at(i + n)`) respectively according to the semantics defined in the language standard.

3.2 A Loop with Task-Level Parallelism

OpenMP 3.0 allows programmers to explicitly create tasks, which enable more parallelization opportunities, especially for algorithms applying independent tasks on non-random accessible data sets, or those using pointer chasing, recursion and so on. It is worthwhile to study how the semantics of high-level abstractions can facilitate parallelization targeting task level parallelism.

An example using the STL list is shown below as a typical candidate for parallelization using an **omp task** directive combined with an **omp single** within an **omp parallel** region:

```
for (std::list<myType>::iterator i = my_list.begin(); i != my_list.end(); i++)
  process(*i);
```

In order to parallelize the loop, a parallelization algorithm has to recognize the following program properties (a conservative case of parallelizable loops):

1. Whether the container supports random access, thus enabling the use of **omp for**; **omp task** is allowed in either case.
2. The elements in the container do not alias or overlap.
3. At most one element accessed via the loop index variable, we refer it as the *current* element, is written within each iteration (no loop carried output dependence among the elements).
4. The loop body does not read elements other than the current element if there is at least one write access to the current element (no loop carried true dependence or antidependence among the elements).
5. There are no other loop carried dependencies caused by variable references other than accessing the elements in the container.

A parallelization algorithm can significantly benefit from the known semantics of standard and user-defined high-level abstractions when dealing with a target mentioned above. It is essential that individual iterations of the loop be independent, substantial analysis is required to verify this. For instance, STL lists do not support random access. Knowing the usage of iterators will help identifying the loop index variable of non-integer types and is critical to recognize the reference to the current element by iterator dereferencing. Element accesses using other than dereferencing the index iterator, such as `front()` and `back()` can be conservatively treated as accesses to non-current elements. Many standard and custom functions have well-defined side effects on both function parameters and/or global variables. Therefore compilers can skip costly side effect analysis for those functions, such as `size()` and `empty()` for STL containers. Domain-specific knowledge can even be used to ensure the uniqueness of elements within a container to be processed as an alternative to conventional alias and pointer analysis. For example, a list of C function definitions returned by a ROSE AST query function has unique and non-overlapping elements.

3.3 A Domain-Specific Tree Traversal

We discuss a specific example from a static analysis tool, namely Compass [10], which is a ROSE-based framework for writing static code analysis tools to detect software defects or bugs. A typical Compass checker's kernel is given in Fig. 2. It is a visitor function to detect any error-prone usage of relational comparison, including $<$, $>$, $\leq$, and $\geq$, on pointers (MISRA Rule 5-0-18 [11]). A recursive tree traversal function walks an input code's AST and invokes the visitor function on each node. Once a potential defect is found, the AST node is stored in a list (`output`) for later display. Most functions (information retrieval functions like `get_*()` and type casting functions like `isSg*()`) used in the function body have read-only semantics.

```
 1  void CompassAnalyses::PointerComparison::Traversal::visit(SgNode* node)
 2  {
 3    SgBinaryOp* bin_op = isSgBinaryOp(node);
 4    if (bin_op)
 5    {
 6      if (isSgGreaterThanOp(node) || isSgGreaterOrEqualOp(node) ||
 7          isSgLessThanOp(node) || isSgLessOrEqualOp(node))
 8      {
 9        SgType* lhs_type = bin_op->get_lhs_operand()->get_type();
10        SgType* rhs_type = bin_op->get_rhs_operand()->get_type();
11        if (isSgPointerType(lhs_type) || isSgPointerType(rhs_type))
12          output->addOutput(bin_op);
13      }
14    }
15  }
```

Fig. 2. A Compass checker's kernel

Even with ideal side effect analysis and alias analysis, a conventional parallelization algorithm will still have trouble in recognizing the kernel as an independent task. The reason is that the write access (line 12) to the shared list will cause an output dependence among different threads, which prevents possible parallelization. However, the kernel's semantics imply that the order of the write accesses does not matter, which make this write access suitable to be protected using **omp critical**. Communicating such semantics to compilers is essential to eliminate the output dependence after adding the synchronization construct.

Another piece of semantic knowledge will enable an even more dramatic optimization. The AST traversal used by Compass checkers does not care about the order of nodes being visited. So it is semantically equal to a loop over the same AST nodes. The AST nodes are stored in memory pools, as in most other compilers [12]. The memory pools in ROSE are implemented as arrays of each type of IR node stored consecutively. Converting a recursive tree traversal into a loop over the memory pools is often beneficial due to better cache locality and less function call overhead. The loop is also more friendly to most analyses and optimizations than the original recursive function call; and importantly to this paper, can be automatically parallelized. In a more aggressive optimization, the types of IR nodes analyzed by the checker can be identified and only the relevant memory pools will be searched.

4 A Semantic-Aware Parallelizer

We design a parallelizer using ROSE to automatically parallelize target loops and functions by introducing either **omp for** or **omp task**, and other required OpenMP directives and clauses. It is designed to handle both conventional loops operating on primitive arrays and modern applications using high-level abstractions. The parallelizer uses the following algorithm:

1. Preparation and Preprocessing
   (a) Read a specification file for known abstractions and semantics.
   (b) Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
   (c) Normalize loops, including those using iterators.
   (d) Find candidate array computation loops with canonical forms (for **omp for**) or loops and functions operating on individual elements (for **omp task**).
2. For each candidate:
   (a) Skip the target if there are function calls without known semantics or side effects.
   (b) Call dependence analysis and liveness analysis.
   (c) Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
   (d) Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.
   (e) Insert the corresponding OpenMP constructs if no dependencies remain.

The key idea of the algorithm is to capture dependencies within a target and eliminate them later on as much as possible based on various rules. Parallelization is safe if there are no remaining dependencies. Semantics of abstractions are used in almost each step to facilitate the transformations and analyses, including recognizing function calls as variable references, identifying the current element being accessed, and ensuring if there are constraints for the ordering of write accesses to shared variables.

The custom transformation for optimizing the Compass checkers is trivial to implement in ROSE since the Compass checkers are derived from an AST traversal class to implement its capability of AST traversal. ROSE already provides AST traversal classes using either recursive tree traversal or loops over memory pools. Changing the checkers' superclass will effectively change the traversal method. Similar to other work [3], our variable classification is largely based on the classic live variable analysis and idiom recognition analysis to identify variables that could be classified as **private**, **firstprivate**, **lastprivate**, and **reduction**.

We give more details of the parallelizer and its handling of high-level abstractions in the following subsections.

4.1 Recognizing High-Level Abstractions and Semantics

ROSE uses a high-level AST which permits the high fidelity representation of both standard and user-defined abstractions in their original source code forms without loss of precision. As a result, program analyses have access to the details of high-level abstraction usage typically lost in a lower level IR. The context of those abstractions can be combined with their known semantics to provide fundamentally more information than could be known from static analysis alone.

Although semantics of standard types and operations can be directly integrated into ROSE to facilitate parallelization, a versatile interface is still favorable to accommodate semantics of user-defined types and functions. As a prototype implementation, we extend the annotation syntax proposed by [13] to manually prepare the specification file representing the knowledge of known types and semantics. A future version of the file will be expressed in C++ syntax to facilitate handling.

The original annotation syntax was designed to allow conventional serial loop optimizations to be applied on user-defined array classes. As a result, it only contains annotation formats for array classes to indicate if the classes are arrays (array) and their corresponding member access functions for array size (**length**()) and elements(**element**()). It also allows users to explicitly indicate read (**read**), written (**modify**), and aliased (**alias**) variables for class operations or functions to complement compiler analysis. We have extended the syntax to accept C++ templates in addition to classes. In particular, **is_fixed_sized_array** is used instead of array to make it clear that a class or template has a set of operations which conform to the semantics of a fixed-sized array, not just any array. Although standard or user-defined high level array abstractions may support some size changing operations such as resize(), those non-conforming operations are not included in the specification file and will be treated as unknown function calls. The semantic-aware parallelizer will safely skip the loop containing such function calls as shown in our algorithm. New semantic keywords have also been introduced to express knowledge critical to parallelization, such as **overlap**, **unique**, and **order_independent**.

An example specification file is given in Fig. 3. It contains a list of qualified names for classes or instantiated class templates with array-like semantics, and their member functions for element access, size query, and other operations preserving the relevant semantics. We also specify side effects of known functions, uniqueness of returned data sets, order-independent write accesses, and so on.

4.2 Dependence Analysis

We generate dependence relations for both eligible loop bodies and function bodies to explore the parallelization opportunities. We compute all dependence relations between every two statements s_1 and s_2, including the case when s_1 is equal to s_2, within the target loop body or function body. Each dependence relation is marked as local or thread-carried (either loop-carried and task-carried).

The foundation of the analysis is the variable reference collection phase, in which all variable references from both statements are collected and categorized into read and write variable sets. In addition to traditional scalar and array references, each member function call returning a C++ reference type is checked against the known high-level abstractions and semantics to see if it is

```
class std::vector<MyType> {
  alias none; overlap none; //elements are alias-free and non-overlapping
  is_fixed_sized_array { //semantic-preserving functions as a fixed-sized array
    length(i) = {this.size()};
    element(i) = {this.operator[](i); this.at(i);};
 };
};
void my_processing(SgNode* func_def) {
  read{func_def}; modify {func_def}; //side effects of a function
}
std::list<SgFunctionDef*> findCFunctionDefinition(SgNode* root){
  read {root}; modify {result};
  return unique; //return a unique set
}
void Compass::OutputObject::addOutput(SgNode* node){
  //order-independent side effects
  read {node}; modify {Compass::OutputObject::outputList<order_independent>};
}
```

Fig. 3. A semantics specification file

semantically equivalent to a subscripted element access of an array-like object. An internal function, is_array(), is used to resolve the type of the object implementing the member function call and compare it to the list of known array types as given in the specification file. If the resolved type turns out to be an instantiated template type, its original template declaration is used for the type comparison instead. Consequently, is_element_access() is applied to the function call to check for an array element access and obtain its subscripts. Read and write variable sets of other known functions are also recognized and the affected variables are collected.

After that, a dependence relation is generated for each pair of references, r_1 from s_1's referenced variable set and r_2 from s_2's, if at least one of the references is a write access and both of them refer to the same memory location based on their qualified variable names or the alias information in the specification file. For array accesses within canonical loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables. The details of the array dependence analysis can be found in [14].

5 Preliminary Results

As this work is an ongoing project (the current implementation is released with the ROSE distribution downloadable from our website [6]), we present some preliminary results in this section. Several sequential kernels in C and C++ were chosen to test our automatic parallelization algorithm on both primitive types and high-level abstractions. They include a C version Jacobi iteration converted from [15] operating on a 500×500 double precision array, a C++ vector 2-norm distance calculation ($\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$) on 100 million elements, and a Compass checker (shown in Fig. 2 for MISRA Rule 5-0-18 [11]) applied on a ROSE source file (Cxx_Grammar.C) with approximately 300K lines of code. The generated OpenMP versions were compiled using our own OpenMP translator, which is a ROSE-based OpenMP 2.5 implementation targeting the Omni OpenMP runtime

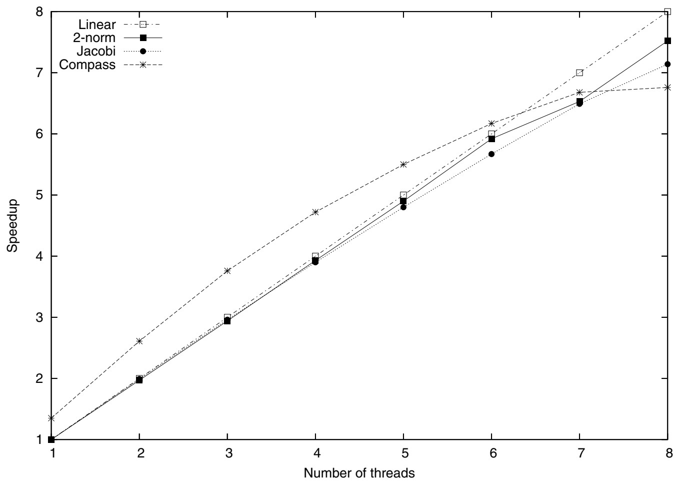


Fig. 4. Speedup of the three example programs after parallelization

library [16]; thus, we do not have performance results for task parallelism (we are currently working on an OpenMP 3.0 implementation and we will also use other OpenMP 3.0 compilers as the backend compiler in the future). GCC 4.1.2 was used as the backend compiler with optimization disabled; optimization is not relevant because we are only showing that our algorithm can extract parallelism from high-level abstractions. We ran the experiments on a Dell Precision T5400 workstation with two sockets, each a 3.16 GHz quad-core Intel Xeon X5460 processor, and 8 GB memory.

Fig. 4 gives speedup of all the three test kernels after domain-specific optimization (optional) and parallelization compared to their original sequential executions. The results proved the efficiency of the semantic-driven optimization of replacing the tree traversal with a loop iteration for the Compass checker: a performance improvement of 35% of the one thread execution compared to the original sequential execution. Our algorithm was also able to capture the parallelization opportunities associated with both primitive data types and high-level abstractions. All tests showed near-linear speedup except for the Compass checker. The critical section within the checker's parallel region made a linear speedup impossible when 7 and 8 threads were used. More dramatic performance improvements can be obtained if only the relevant memory pools are searched but this step is not yet automated in our implementation.

6 Related Work

Numerous research compilers have been developed to support automatic parallelization. We only mention a few of them due to the page limit. For example, the Vienna Fortran compiler (VFC) [17] is a source-to-source parallelization system for an optimized version of High Performance Fortran. The Polaris compiler [2] is mainly used for improving loop-level automatic parallelization. The SUIF compiler [18] was designed to be a parallelizing and optimizing compiler supporting multiple languages. However, to the best of our knowledge, current research parallelizing compilers largely focus on Fortran and/or C applications. Commercial parallelizing compilers like the Intel C++/Fortran compiler [3] also use OpenMP internally as a target for automatic parallelization. Our work in ROSE aims to complement existing compilers by providing a source-to-source, extensible parallelizing compiler infrastructure targeting modern object-oriented applications using both standard and user-defined high-level abstractions.

Several papers in the literature present parallelization efforts for C++ Standard Template Library (STL) or generic libraries. The Parallel Standard Template Library (PSTL) [19] uses parallel iterators and provides some parallel containers and algorithms. The Standard Template Adaptive Parallel Library (STAPL) [20] is a superset of the C++ STL. It supports both automatic parallelization and user specified parallelization policies with several major components for containers, algorithms, random access range, data distribution, scheduling and execution. GCC 4.3's runtime library (libstdc++) provides an experimental parallel mode, which implements an OpenMP version of many C++ standard library algorithms [21]. Kambadur et al. [22] proposes a set of language extensions to better support C++ iterators and function objects in generic libraries. However, all library-based parallelization methods require users to make sure that their applications are parallelizable. Our work automatically ensures the safety of parallelization based on semantics of high-level abstractions and compiler analyses.

Some previous research has explored code analyses and optimizations based on high-level semantics. STLlint [23] performs static checking for STL usage based on symbolic execution. Yi and Quinlan [13] developed a set of sophisticated semantic annotations to enable conventional sequential loop optimizations on user-defined array classes. Quinlan et al. [24, 25] presented the parallelization opportunities solely using the high-level semantics of A++/P++ libraries and user-defined C++ containers without using dependence analysis. This paper combines both standard and user-defined semantics with compiler analyses to further broaden the applicable scenarios of automatic parallelization. We also consider the new OpenMP 3.0 features and domain-specific optimizations.

7 Conclusions and Future Work

In this paper, we have explored the impact of high-level abstractions on automatic parallelization of C++ applications and designed a parallelization algorithm to take advantage of the capability of the ROSE source-to-source compiler infrastructure and the known semantics of both standard and user-defined abstractions. Though only three representative cases have been examined, our approach is very generic so that additional STL or user-defined semantics which are important to parallelization can be discovered and incorporated into our implementation. Our work demonstrates that semantic-driven parallelization is a very feasible and powerful approach to capture more parallelization opportunities than conventional parallelization methods for multicore architectures. Our approach can also be seamlessly integrated with conventional analysis-driven parallelization algorithms as a significant complement or enhancement.

In the future, we will apply our method on large-scale C++ applications to recognize and classify more semantics which can be critical to parallelization. We are planning to extend our work to support applications using more complex and dynamic control flows such as pointer chasing and use more OpenMP construct types. Further work also includes investigating the impact of polymorphism used in C++ applications, exploring the interaction between the automatic parallelization and conventional loop transformations, and leveraging semantics for better OpenMP optimizations as well as correctness analyses.

References

1. OpenMP Architecture Review Board: The OpenMP specification for parallel programming (2008), `http://www.openmp.org`
2. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with Polaris. Computer 29(12), 78–82 (1996)
3. Bik, A., Girkar, M., Grey, P., Tian, X.: Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. Intel. Technology Journal 5 (2001)
4. Johnson, S.P., Evans, E., Jin, H., Ierotheou, C.S.: The ParaWise Expert Assistant — widening accessibility to efficient and scalable tool generated OpenMP code. In: Chapman, B.M. (ed.) WOMPAT 2004. LNCS, vol. 3349, pp. 67–82. Springer, Heidelberg (2005)
5. Liao, S.W., Diwan, A., Robert, P., Bosch, J., Ghuloum, A., Lam, M.S.: SUIF Explorer: an interactive and interprocedural parallelizer. In: PPoPP 1999: Proceedings of the seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp. 37–48. ACM Press, New York (1999)
6. Quinlan, D.J., et al.: ROSE compiler project, `http://www.rosecompiler.org/`
7. Edison Design Group: C++ Front End, `http://www.edg.com`
8. Rasmussen, C., et al.: Open Fortran Parser, `http://fortran-parser.sourceforge.net/`
9. Bodin, F., et al.: Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. In: Proceedings of the Second Annual Object-Oriented Numerics Conference (1994)
10. Quinlan, D.J., et al.: Compass user manual (2008), `http://www.rosecompiler.org/compass.pdf`
11. The Motor Industry Software Reliability Association: MISRA C++: 2008 Guidelines for the use of the C++ language in critical systems (2008)
12. Cooper, K., Torczon, L.: Engineering a Compiler. Morgan Kaufmann, San Francisco (2003)
13. Yi, Q., Quinlan, D.: Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In: The 17th International Workshop on Languages and Compilers for Parallel Computing, LCPC (2004)
14. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann, San Francisco (2001)

15. Robicheaux, J., Shah, S. (1998), `http://www.openmp.org/samples/jacobi.f`
16. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP compiler for an SMP cluster. In: the 1st European Workshop on OpenMP (EWOMP 1999), September 1999, pp. 32–39 (1999)
17. Benkner, S.: VFC: The Vienna Fortran Compiler. Scientific Programming 7(1), 67–81 (1999)
18. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.A.M., Tjiang, S.W., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices 29(12), 31–37 (1994)
19. Johnson, E., Gannon, D., Beckman, P.: HPC++: Experiments with the Parallel Standard Template Library. In: Proceedings of the 11th International Conference on Supercomputing (ICS 1997), pp. 124–131. ACM Press, New York (1997)
20. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N.M., Rauchwerger, L.: STAPL: An adaptive, generic parallel C++ library. In: Languages and Compilers for Parallel Computing (LCPC), pp. 193–208 (2001)
21. Singler, J., Konsik, B.: The GNU libstdc++ parallel mode: software engineering considerations. In: IWMSE 2008: Proceedings of the 1st international workshop on Multicore software engineering, pp. 15–22. ACM, New York (2008)
22. Kambadur, P., Gregor, D., Lumsdaine, A.: OpenMP extensions for generic libraries. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 123–133. Springer, Heidelberg (2008)
23. Gregor, D., Schupp, S.: STLlint: lifting static checking from languages to libraries. Softw. Pract. Exper. 36(3), 225–254 (2006)
24. Quinlan, D.J., Schordan, M., Yi, Q., de Supinski, B.R.: Semantic-driven parallelization of loops operating on user-defined containers. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 524–538. Springer, Heidelberg (2004)
25. Quinlan, D., Schordan, M., Yi, Q., de Supinski, B.: A C++ infrastructure for automatic introduction and translation of OpenMP directives. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 13–25. Springer, Heidelberg (2003)

Scalability Evaluation of Barrier Algorithms for OpenMP

Ramachandra Nanjegowda[1], Oscar Hernandez[1], Barbara Chapman[1], and Haoqiang H. Jin[2]

[1] Computer Science Department
University of Houston, Houston, Texas 77004, USA
rchakena@uh.edu, oscar@cs.uh.edu, chapman@cs.uh.edu
[2] NASA Ames Research Center, Moffett Field, CA 94035
haoqiang.jin@nasa.gov

Abstract. OpenMP relies heavily on barrier synchronization to coordinate the work of threads that are performing the computations in a parallel region. A good implementation of barriers is thus an important part of any implementation of this API. As the number of cores in shared and distributed shared memory machines continues to grow, the quality of the barrier implementation is critical for application scalability. There are a number of known algorithms for providing barriers in software. In this paper, we consider some of the most widely used approaches for implementing barriers on large-scale shared-memory multiprocessor systems: a "blocking" implementation that de-schedules a waiting thread, a "centralized" busy wait and three forms of distributed "busy" wait implementations are discussed. We have implemented the barrier algorithms in the runtime library associated with a research compiler, OpenUH. We first compare the impact of these algorithms on the overheads incurred for OpenMP constructs that involve a barrier, possibly implicitly. We then show how the different barrier implementations influence the performance of two different OpenMP application codes.

1 Introduction

OpenMP programs typically make repeated use of barriers to synchronize the threads that share the work they contain. Implicit barriers are required at the end of parallel regions; they are also required at the end of worksharing constructs unless explicitly suppressed via a *NOWAIT* clause. Explicit barriers may be inserted elsewhere by the application developer as needed in order to ensure the correct ordering of operations performed by concurrently executing, independent threads. The OpenMP implementation will therefore need to include barrier synchronization operations: these are typically an important part of the runtime library, whose routines are inserted into OpenMP code during compilation and subsequently invoked during execution. Given the importance of this construct in the OpenMP API, a good barrier implementation is essential.

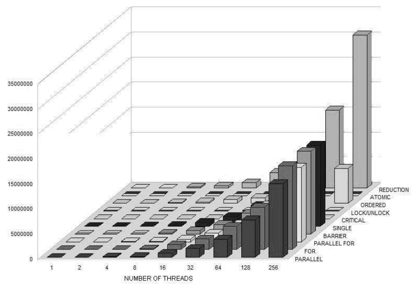
In OpenMP 3.0, threads waiting at barriers are permitted to execute other tasks. Hence eliminating barrier contentions is essential to free up threads or improve work stealing. As the number of threads in parallel regions steadily grows, and the memory hierarchies in the cores and parallel machines become more complex, the scalability of the implementation becomes increasingly important. OpenMP implementations should provide a choice of barrier implementations for different environments and architectures with complex memory hierarchies and interconnect topologies.

Barrier algorithms are generally considered to fall into two classes: those based upon a blocking construct that de-schedules a waiting thread, and those relying on a busy-wait construct, in which threads loop on a shared variable that needs to be set or cleared before they can proceed. The primary disadvantage of scheduler-based blocking is that the overhead of scheduling may exceed the expected wait time. The scheduling overhead we are referring here is the overhead involved in switching the thread back and forth when it is blocked. The operating system scheduler will switch the blocked thread and reliquish the processor, for any other thread which is ready to execute. On the other hand, the typical implementation of busy-waiting introduces large amounts of memory and interconnect contention which causes performance bottlenecks. Hence enhanced versions of a busy-wait barrier implementation, generally known as distributed busy-wait , have been devised. The key to these algorithms is that every processor spins on a separate locally-accessible flag.

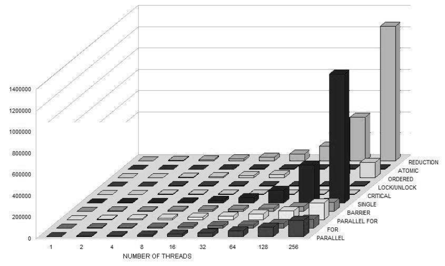
As part of an effort to investigate ways in which OpenMP and its implementations may scale to large thread counts, we have begun to study a variety of strategies for accomplishing the most expensive synchronization operations implied by this API, including barriers and reductions. We believe that the existing set of features for expression of the concurrency and synchronization within an application must be enhanced in order to support higher levels of shared memory concurrency, but that a careful implementation of existing features may go some of the way to improving the usefulness of this programming model on large shared-memory machines. The aim of the work described here is to gain insight into the behavior of different barrier algorithms in the OpenMP context in order to determine which of them is most appropriate for a given scenario. Our overall goal is to provide an OpenMP library which will adapt to deliver the most suitable implementation of a barrier based on the number of threads used, the architecture (memory layout/interconnect topology), application and possibly system load.

2 OpenUH Implementation of OpenMP

The experimental results described here are based upon barrier algorithms implemented in the runtime library associated with the OpenUH compiler. OpenUH [1], a branch of the Open64 [2] compiler suite, is an optimizing and portable open-source OpenMP compiler for C/C++ and Fortran 95 programs. The suite is based on SGI's Pro64 compiler which has been provided to the community as open source.



(a)



(b)

Fig. 1. (a) The performance of EPCC syncbench compiled using OpenUH. (b) The performance of EPCC syncbench compiled using Intel compiler.

At the time of writing, OpenMP 2.5 is supported along with a partial implementation of OpenMP 3.0; the compiler's OpenMP runtime library is open source. It targets the Itanium, Opteron, and x86 platforms, for which object code is produced, and may be used as a source-to-source compiler for other architectures using the intermediate representation (IR)-to-source tools.

OpenUH translates OpenMP to an internal runtime API, which provides the internal data structures and thread management needed to implement OpenMP constructs. The OpenMP runtime uses the Posix Threads API for thread creation, thread signals, and locks with the goal of providing a portable OpenMP runtime implementation. The initial barrier implementation in OpenUH was based on a very straightforward centralized blocking barrier algorithm, described in the next section, and which was known to have poor scalability.

In Figure 1, we show the overheads of the different OpenMP constructs as the number of processors increases on a SGI Altix 3700 using the EPCC microbenchmarks. The graph on the left gives the overheads in milliseconds reported for OpenUH, and those of the Intel 10.1 compiler are given on the right. A quick inspection of these results shows that there are OpenMP operations which are significantly more expensive than others in both implementations: these include reductions, single constructs, parallel/parallel for and barriers. It is important to note that the overhead of such constructs depends on the way the barrier is implemented, since they are either the barrier or they contain an implicit barrier.

3 Approaches to Implementing a Barrier

In this section we describe several different implementations of the barrier construct that we considered for this work. All of them appear in the literature, and have been implemented in different parallel languages and libraries, including MPI, UPC, CoArray Fortran and Global Arrays. Our interest was to evaluate them with regard to their usefulness for OpenMP and to determine whether one or more of them provided superior performance. We use a pseudo code representation to describe the algorithms.

3.1 Centralized Blocking Barrier

This barrier implementation is based on a a single thread counter, a mutex and a condition variable. The counter keeps track of the number of threads that have reached a barrier. If the counter is less than the total number of threads, the threads do a condition wait. The last thread to reach this barrier wakes up all the threads using condition broadcast. The mutex is used to synchronize access by the different threads to the shared counter.

The need to acquire the mutex lock is the main bottleneck in this algorithm, as all the threads compete to acquire it when they receive the broadcast signal. An alternative implementation uses multiple condition variables and mutexes to relieve this. A pair of threads will then use a particular set of mutex and condition variables, which will provide somewhat better performance than is obtained by using a single mutex and condition variable.

Under both these strategies, the scheduling overhead (the overhead of context switching the blocked thread) is far more expensive than the typically expected barrier wait time, and hence this simplistic algorithm is not an efficient way to implement a barrier where thread blocking is not needed. However this algorithm makes sense when we oversubscribe threads to cores (i.e. several threads share a core), because it frees up CPU resources allowing other threads to obtain CPU time.

3.2 Centralized Barrier

In a centralized barrier algorithm, each processor updates a counter to indicate that it has arrived at the barrier and then repeatedly polls a flag that is set

```
// Shared data.
barrier_lock // pthread mutex lock
barrier_cond // pthread condition variable
barrier_count // Number of threads not yet arrived
barrier_flag // To indicate all arrived

procedure blocking_barrier
  barrier_flag=team->barrier-flag
  new_count = atomic_incr(barrier_count)

  if(new_count==team_size)
    team->barrier_count=0
    team->barrier_flag = barrier_flag ^ 1
    pthread_cv_broadcast()
  else
    pthread_mutex_lock()
    while(barrier_flag==team->barrier_flag)
       pthread_cv_wait()
    pthread_mutex_unlock()
```

(a) Blocking Barrier Algorithm

```
// Shared data.
barrier_count // not yet arrived
barrier_flag // indicate all have arrived
team_size // number of threads

procedure central_barrier
  barrier_flag=team->barrier-flag
  new_count = atomic_inc(barrier_count)

if (new_count == team_size)
  team->barrier_count = 0
  team->barrier_flag = barrier_flag ^ 1
else
 while (team->barrier_flag == barrier_flag)
```

(b) Centralized Barrier Algorithm

when all threads have reached the barrier. Once all threads have arrived, each of them is allowed to continue past the barrier. The flag can be a sense reversal flag, to prevent intervention of adjacent barrier operations.

This implementation uses a small amount of memory, is simple to implement, and could be good if cores share small fast caches (typically L2).

The potential drawback of centralized barriers is that the busy waiting to test the value of the flag occurs on a single, shared location. As a result, centralized barrier algorithms cannot be expected to scale well to large numbers of threads. Our experiments (see Section 4 below) confirm this. However, because of the small amount of memory used, it may be a contender where cores share cache lines, as it keeps the rest of the cache almost intact.

3.3 Dissemination Barrier

This barrier implementation technique derives its name from the way in which it disseminates information among a set of threads. Each thread spins around a variable dedicated to it, and signaled by another thread. Each thread goes through log(N) rounds where N is the number of threads. At the end of rounds, the thread knows that the other threads in the system have reached the barrier and it is good to proceed to the next barrier episode.

In step k, thread i signals thread (i+2k) mod P. For each step, we use alternate sets of variables to prevent interference in consecutive barrier operations. This algorithm also uses sense reversal to avoid resetting variables after every barrier. The dissemination barrier has shorter critical as compared to other algorithms. Our experiments (see Table 1) shows that this algorithm gives the best performance upto 16 threads. It also gives best performance for the barrier contruct on multicore platform (see Table 2).

```
// Shared data:
int P // number of threads
struct node {
boolean flag[2]
struct node *partner
}
node nodes[P][logP] // array of nodes

// Private data for each thread:
volatile int parity
volatile int sense

// intializes each thread
// to its partner
procedure dissem_init() {
 for (i = 0; i < P; i++) {
  d = 1;
  for (r = 0; r < logP; r++) {
nodes[i][j].partner=nodes[(i+d) % P][j];
d = 2*d;
}
 }
}
```

(a) Dissemination Barrier Initialization

```
procedure dissem_barrier {
 i = thread_id;
 sense = thread_private->sense
 parity = thread_private->parity
 for ( r = 0; r < logP; r++) {
  nodes[i][r].partner.flag[parity]= sense;
  while (nodes[i][r].flag[parity] != sense)
 }

 if(parity==1)
   thread_private->sense = sense^1;
 thread_private->parity=1-parity;
}
```

(b) Dissemination Barrier Algorithm

3.4 Tree Barrier

In this method, each thread is assigned to a unique tree node which is linked into an arrival tree by a parent link and into the wakeup tree by a set of child links. The parent notifies each of its children by setting a flag in the nodes corresponding to them. The child in turn sets a flag in the parent node to signal its arrival at the barrier.

The data structures for the tree barrier is initialized such that each node's parent flag variable points to the appropriate *childnotready* flag. The *child_notify* variable points to the *wakeup_sense* variable. The *havechild* flag indicates whether a particular node has children or not. During a barrier phase, a thread tests to see if the *childnotready* flag is clear for each of its children before reinitializing them to next barrier. After all children of a node have arrived, the *childnotready* flag is cleared. All threads other than root spins on their local *wakeup_sense* flag. At each level, a thread releases all its children before leaving the barrier and thus eventually the barrier is complete.

3.5 Tournament Barrier

The Tournament barrier algorithm is similar to a tournament game. Two threads play against each other in each round. The loser thread sets the flag on which the

```
// Shared data:
typedef struct {
        volatile boolean *parentflag;
        boolean *child_notify[2];
        volatile long havechild;
        volatile long  childnotready;
        boolean wakeup_sense;
} treenode;
treenode shared_array[P];

Private data for each thread:
  volatile int parity;
  volatile boolean sense;
  treenode *mynode;

procedure tree_barrier
 vpid=thread_id;
 treenode *mynode_reg;
 mynode_reg = cur_thread->mynode;

 while (mynode_reg->childnotready);

 mynode_reg->childnotready = mynode_reg->havechild;
 *mynode_reg->parentflag = False;

 if (vpid)
   while(mynode_reg->wakeup_sense != cur_thread->sense);

 *mynode_reg->child_notify[0] = cur_thread->sense;
 *mynode_reg->child_notify[1] = cur_thread->sense;
 cur_thread->sense ^= True;
}
```

(a) Tree Barrier Algorithm

```
// Shared data:
struct round_t
{
boolean *opponent
int role // WINNER or LOSER
boolean flag
}
round_t rounds[P][logP]

// Private data for each thread:
boolean sense
int parity
round_t *myrounds

procedure tour_barrier
round_t round=current_v_thread->myrounds;
for(;;) {
  if(round->role & LOSER) {
     round->opponent->flag = sense;
while (root_sense != sense);
    break;
   }
   else if (round->role & WINNER)
   while (round->flag != sense);
   else if (round->role & ROOT) {
     while (round->flag != sense);
     champion_sense = sense;
     break;
   }
 round++;
}
```

(b) Tournament Barrier Algorithm

the winner is busy waiting. Then the loser thread waits for the global champion flag to be set, where as the winners, play against each other in next round. The overall winner becomes the champion and notifies all losers about the end of barrier.

The complete tournament barrier requires log N “rounds”, where N is the number of threads. The threads begin at the leaves of the binary tree and at each step, one thread continues up to the tree to the next round of the tournament. The WINNER and LOSER at each stage is statically determined during initialization. In round k, thread i sets a flag of thread j, where i = 2^k, and j = $(i - 2)^k$. The LOSER thread i drops out and busy waits on a global flag while the WINNER thread j, participates in the next round of the tournament.

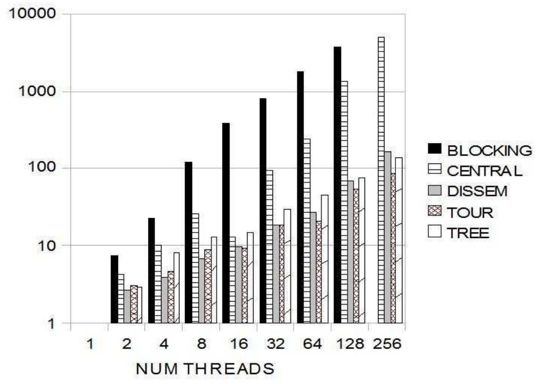
4 Performance Measurements

We have tested the barrier implementations described above on several different platforms. Experiments were performed on Cobalt, NCSA’s SGI 3700 Altix, consisting of two systems each with 512 Intel 1.6 Ghz Itanium2 Madison processors running SUSE 10.2 (see `http://www.teragrid.org/` for a detailed description of the system). The experiments up to 512 threads was run on Columbia, NASA’s SGI 4700 Altix, consisting of 1024 dual-core Intel 1.6 Ghz Itanium2 Montecito processors running SUSE Linux Enterprise Operating system (see `http://www.nas.nasa.gov/` for a detailed description of the system). Other experiments were conducted on a Sun Fire X4600 with eight dual core AMD Opteron 885 processors and a Fujitsu-Siemens RX600S4/X system with four Intel Xeon E7350 quad core processors located a Aachen University.

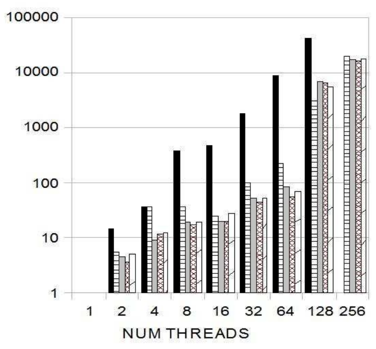
4.1 EPCC Microbenchmark Results

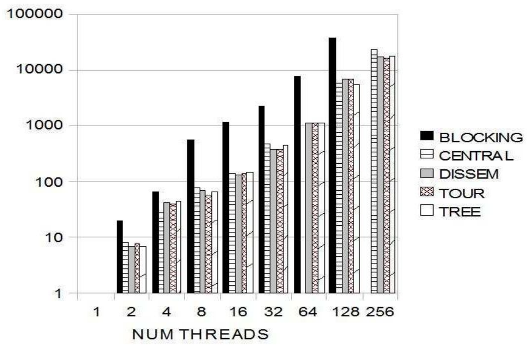
We implemented each of the five algorithms described above in the OpenUH runtime library and used the corresponding portion of the EPCC microbenchmarks to test the barrier performance they supply. The first diagram (See Fig a) gives the time taken to implement a barrier in microseconds for 2, 4, 16, 32, 64, 128 and 256 threads on SGI 3700. As can be clearly seen, the centralized blocking and centralized barrier algorithm did not perform as well as the tournament and dissemination barrier. This is expected since our test cases didn’t involve oversubscribing threads to cores. The tournament algorithm resulted in the least overhead where the thread count is greater than 16 and the dissemination barrier is best when the number of threads is less than 16. For space reasons, we did not show results of our tests on a Sun Fire X4600 and a Fujitsu-Siemens Intel Xeon with up to 16 threads. In both systems, the dissemination barrier produced the least overhead (See Table 2 for summarized results).

The next microbenchmark tests the time taken to execute a parallel directive, including the barrier which is required at its termination. As before, we show overheads for 2, 4, 16, 32, 64, 128 and 256 threads. Here the results (See Fig b) were similar to the previous barrier case where the tournament barrier produced



(a) BARRIER





(b) PARALLEL (c) REDUCTION

lower overheads for test cases with 16 or more threads. The blocking algorithm continues to perform poorly (and we do not show results above 128 threads).

A barrier is also used as part of the OpenMP reduction implementation. The results (See Fig c)for the reduction operation are more diverse than for the other two OpenMP constructs (Barrier and Parallel). The tree implementation produced the least overheads with 2 and 128 threads, the centralized barrier performed well on 4 threads, the dissemination barrier worked well for 16 threads, and the tournament implementation worked well on 32, 64 and 256 threads. The blocking algorithm produced the worst results in all cases. On a Sun Fire X4600 and a Fujitsu-Siemens Intel Xeon running the test case with 2, 4, 8 and 16 threads, the tournament barrier produced the best results (See Table 2).

Table 1 summarizes the best algorithms for the different OpenMP constructs based on the EPCC benchmark on SGI 3700 Altix until 256 threads and SGI 4700 Altix for 512 threads. Table 2 summarizes the results on multicore systems (Sun Fire X4600 with eight dual core AMD Opteron 885 processors). It is clear that there is not a single optimal algorithm for all the different OpenMP constructs

Table 1. Best Barrier Algorithms in the EPCC Benchmark on the SGI 3700 Altix until 256 threads and SGI 4700 Altix for 512 threads

Number of Threads	Barrier	Reduction	Parallel
2	dissemination	tree	tournament
4	dissemination	centralized	dissemination
8	dissemination	tournament	tournament
16	tournament	dissemination	tournament
32	tournament	tournament	tournament
64	tournament	tournament	tournament
128	tournament	tree	tournament
256	tournament	tournament	tournament
512	tournament	tournament	tournament

Table 2. Best Barrier Algorithms in the EPCC Benchmark on Sun Fire X4600

Number of Threads	Barrier	Reduction	Parallel
2	dissemination	tournament	tournament
4	dissemination	tournament	tournament
8	dissemination	tournament	tournament
16	dissemination	tournament	tournament

with different numbers of threads and on different platforms. The best barrier implementation depends on the environment (i.e, number of threads, system utilization) and the platform where the application is running. On a multicore system, thread binding also influenced the results. We saw an improvement in performance of these barrier implementations when the threads were bound to cores of the same processor using numactl command. These results show the need for OpenMP runtimes to be able to adaptively choose the best barrier implementation based on all these factors.

4.2 The Performance of ASPCG

We tested the barrier implementation using the Additive Schwarz Preconditioned Conjugate Gradient (ASPCG) kernel up to 128 threads on the Altix System. The ASPCG kernel solves a linear system of equations generated by a Laplace equation in Cartesian coordinates. The kernel supports multiple parallel programming models including OpenMP and MPI.

Figure 2 shows the timings of the ASPCG kernel using the different barrier implementations. Note that the blocking algorithm does not scale after 32 threads. This is because the wake-up signal to the threads becomes a contention point. All busy-wait algorithms scale, some with better performance than others. Using the dissemination implementation instead of the original barrier implementation in OpenUH, centralized blocking, represents a performance gain (total wall clock time) of 12 times for 128 threads. Table 3 shows a summary of the best barrier implementations for ASPCG on different numbers of threads.

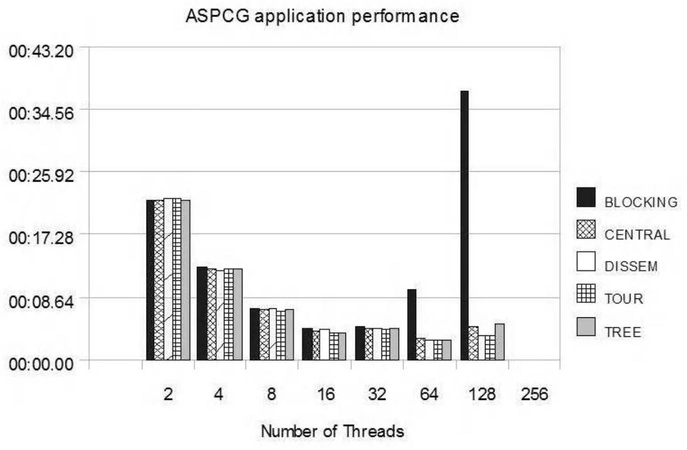


Fig. 2. Timings for the ASPCG kernel with different barrier implementations

Table 3. Best Barrier Algorithms for ASPCG and GenIDLEST

Number of Threads	ASPCG	GenIDLEST
2	tournament	blocking
4	dissemination	blocking
16	tournament/tree	dissemination
32	tournament	tournament
64	tournament	–
128	dissemination	–

4.3 The Performance of GenIDLEST

Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST) solves the incompressible Navier-Stokes and energy equations and is a comprehensive and powerful simulation code with two-phase dispersed flow modeling capability, turbulence modeling capabilities, and boundary conditions to make it applicable to a wide range of real world problems. It uses an overlapping multi-block body-fitted structured mesh topology in each block combining it with an unstructured inter-block topology. The multiblock approach provides a basic framework for parallelization, which is implemented by SPMD parallelism using MPI, OpenMP or a hybrid MPI/OpenMP. GenIDLEST uses OpenMP *for* and *reduction* constructs extensively in its code. In GenIDLEST, the centralized blocking implementations works better than the rest of the barrier implementations for 2 and 4 threads. Table 3 shows a summary of the best barrier implementations for different numbers of threads. It is clear here that the choice of a good barrier implementation can be application dependent. The use of the tournament barrier algorithm instead of a centralized blocking one represented a performance gain in total wall clock time of 35% on 32 threads.

5 Conclusions and Future Work

We have presented the impact of a number of different barrier implementations, including a centralized blocking algorithm and several kinds of busy wait algorithms, on the overheads of OpenMP constructs.

We implemented the barrier algorithms in OpenUH and obtained a significant reduction in overheads for barriers, and constructs that include them, using distributed busy-wait approaches. The ASPCG and GENIDLEST applications were compiled using the enhanced OpenUH system, also with noticeable performance improvements under the new busy-wait algorithms. In general, the performance of a given barrier implementation is dependent on the number of threads used, the architecture (memory layout/interconnect), application and possibly system load. An OpenMP runtime library should therefore, we believe, adapt to different barrier implementations during runtime.

Our future work includes further testing of these algorithms on larger systems (in particular, the Altix 4700 deployed at Nasa Ames with 2048 cores) and on other applications. We also plan to explore the use of similar enhancements to further improve the performance the reduction operation (i.e. updates on the reduction variable). Also as OpenMP 3.0 becomes available in OpenUH we would like to evaluate how these algorithms affect the performance of the tasking feature, especially the untied tasks and the environment variable `OMP_WAIT_POLICY`.

Acknowledgments

We want to thank Dr. Edgar Gabriel for providing us useful information about MPI barrier implementations in OpenMPI and Cody Addison for providing us preliminary results that motivated this work. We also want to thank the Center for Computing and Communication, at RWTH Aachen University for letting us use their cluster. This work is supported by NSF grants CCF-0833201 and CCF-0702775 (see http://www2.cs.uh.edu/ hpctools/darwin/). The pseudocode published by the Computer Science Department of the University of Rochester was adapted for the work described in this paper.

References

[1] Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: Openuh: An optimizing, portable openmp compiler. In: 12th Workshop on Compilers for Parallel Computers (January 2006)

[2] Open64 (2005), `http://open64.sourceforge.net`

Use of Cluster OpenMP with the *Gaussian* Quantum Chemistry Code: A Preliminary Performance Analysis

Rui Yang, Jie Cai, Alistair P. Rendell, and V. Ganesh

Department of Computer Science
College of Engineering and Computer Science
The Australian National University,
Canberra, ACT 0200, Australia
`{Rui.Yang,Jie.Cai,Alistair.Rendell,`
`Ganesh.Venkateshwara}@anu.edu.au`

Abstract. The Intel Cluster OpenMP (CLOMP) compiler and associated runtime environment offer the potential to run OpenMP applications over a few nodes of a cluster. This paper reports on our efforts to use CLOMP with the *Gaussian* quantum chemistry code. Sample results on a four node quad core Intel cluster show reasonable speedups. In some cases it is found preferable to use multiple nodes compared to using multiple cores within a single node. The performances of the different benchmarks are analyzed in terms of page faults and by using a critical path analysis technique.

Keywords: Cluster OpenMP, performance, quantum chemistry code.

1 Introduction

OpenMP is an attractive shared memory parallel programming model that invites incremental parallelization. Traditionally, however, OpenMP programs have been constrained by the number of processors and available memory on the underlying shared memory hardware, as changing these involves significant expense. By contrast for message passing applications running on a cluster it is usually a relatively easy matter to add a few more nodes to a cluster. In this respect the ability to run an OpenMP code over even a few nodes of a cluster, as promised by the Intel Cluster OpenMP (CLOMP) compiler and runtime environment [1], has an immediate attraction.

CLOMP runs over a cluster by mapping the OpenMP constructs to an underlying page based software Distributed Shared Memory (DSM) system [1]. To transfer an application from native OpenMP to CLOMP is in principle straightforward; requiring shared variables to be identified as ‘sharable’ either automatically by the compiler or manually by the programmer placing sharable directives where the variable is declared. Sharable data are placed on sharable pages which are read/write protected using the **`mprotect()`** system call. When an OpenMP thread accesses data on one of these pages a segmentation fault occurs that is caught by the CLOMP runtime library. The library is then responsible for retrieving the relevant memory page from the distributed environment, ensuring that the data is consistent with what is expected given the OpenMP memory consistency model.

Compared to regular OpenMP, the additional overhead of using CLOMP is primarily that associated with the cost of servicing the various segmentation faults. As the latency of any communication between nodes on a cluster is in the order of microseconds, while the clock speed of a typical processor is sub nanosecond, for an application to scale well using CLOMP it will need to perform significant computation between memory consistency points. Ideal CLOMP applications are those that may access large amounts of sharable data but modify only a relatively small amount of this data, and make limited use of thread synchronization. Good examples are applications for doing rendering, data-mining, all kinds of parallel search, speech and visual recognition, and genetic sequencing [2].

In this paper we report on our efforts to adapt parts of the *Gaussian* computation chemistry package [3] for use with CLOMP. In section 2 we give a brief overview of the *Gaussian* code and the parts that we have attempted to use CLOMP with. Section 3 details our benchmark environment and computations, while section 4 discusses the observed performance attempting to rationalize it in terms of the performance model of Cai et al. [4]. Finally section 5 contains our conclusions.

2 The *Gaussian* Program and Its Use with CLOMP

Gaussian is a general purpose computational chemistry package that can perform a variety of electronic structure calculations [3]. It consists of a collection of executable programs or “Links” that are used to perform different tasks. A complete *Gaussian* calculation involves executing a series of Links for different purposes such as data input, electronic integral calculation, self-consistent field (SCF) evaluation etc. Each Link is responsible for continuing the sequence of Links by invoking the **exec()** system call to run the next Link. Not all Links run in parallel; those that run sequentially are mainly responsible for setting up the calculations and assigning symmetry and will usually take only a short time to execute [5].

Two fundamental quantum chemistry methods are Hartree-Fock theory and Density Functional Theory (DFT). In the *Gaussian* code the implementations of both methods are very similar. Specifically in both cases electrons occupy molecular orbitals that are expanded in terms of a set of basis functions (usually atom centered), for which the coefficients are obtained by solving an equation of the form:

$$\mathrm{F}\, C = \varepsilon\, \mathrm{S}\, C \tag{1}$$

where C is a matrix containing the orbital coefficients; S is the overlap matrix with elements that represent the overlap between two basis functions; ε is a diagonal matrix of yet to-be-determined molecular orbital energies, and in the case of Hartree-Fock theory, F is the Fock matrix. The Fock matrix is defined as:

$$F_{\mu\nu} = H_{\mu\nu}^{Core} + \sum_{\lambda\sigma}^{N_{basis}} P_{\lambda\sigma}\left[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\sigma|\lambda\nu)\right] \tag{2}$$

where $H_{\mu\nu}^{Core}$ involves integrals representing the kinetic energy of the electrons and their interactions with the nuclei, and the remaining part of the right side of Eqn. (2) comes from the interactions between electrons. In the latter the two-electron repulsion integrals (ERIs) are defined as:

$$(\mu\nu|\lambda\sigma) = \int d\mathbf{r} \int d\mathbf{r}' \frac{\phi_\mu(r)\phi_\nu(r)\phi_\lambda(r')\phi_\sigma(r')}{|\mathbf{r}-\mathbf{r}'|} \tag{3}$$

and *P* is the density matrix which for a closed shell system is given by:

$$P_{\mu\nu} = 2\sum_m C_{\mu m} C_{\nu m}^* \tag{4}$$

with the summation being over half the number of electrons in the system.

For DFT the equations are essentially the same, except that *F* is now the Kohn-Sham matrix *K*, the second "exchange" term in the summation in Eqn. (2) is dropped, and an additional term is added that involves a numerical integration over all space of a quantity that depends on the electron density. Between HF and DFT there lies a number of so called hybrid schemes that include some HF exchange as well as some of the DFT terms requiring numerical integration.

A HF energy calculation performed using *Gaussian* uses a subroutine called PRISM to calculate the relevant ERIs and form their contribution to the Fock matrix. For DFT calculations a subroutine called PRISMC is used to evaluate the ERIs, while the numerical integration is undertaken by a routine called CALDFT. For hybrid DFT methods PRISM is used together with CALDFT. Collectively these three routines will typically consume over 90% of the total execution time for a HF or DFT energy calculation. For this reason these routines have already been parallelized for shared memory hardware using OpenMP; adapting these routines to run with CLOMP and studying their performance is the major objective of this work.

After the formation of the *F* (or *K*) matrix, the molecular orbital coefficients *C* are determined by solving Eqn. (1). This involves diagonalization of *F* (or *K*). The new molecular orbital coefficients are then put back into Eqn. (4) to form a new density matrix from which a new *F* (or *K*) matrix is computed. This process continues until there is no change in the density matrix between iterations, at which point the interactions between the electrons represents a "self-consistent field". In *Gaussian*, Link 502 is used to perform the SCF computation and evaluate the total energy for a given configuration of the atomic nuclei.

Most of quantum chemistry revolves around atomic configurations where there are no net forces on the individual nuclei. Finding these locations involves evaluating both the energy of the system and its derivative with respect to a displacement of any of the atomic nuclei. In *Gaussian* such force evaluations are performed for HF and DFT methods by using the same three routines (PRISM, PRISMC and CALDFT) but called from Link 703.

The basic strategy for a HF/DFT energy evaluation is as follows:

1. Start the Link using a single process (the master thread) and allocate a large working memory space using the **`malloc()`** system call. Data used in the evaluation of the *F* (or *K*) matrix is subsequently stored into this working space (e.g. the density matrix and orbital shell information). All density matrix related input data gets arranged as a single block, with the length of this "density matrix block" being dependent on the size of the problem (atoms, theoretical method and basis set size).
2. In the routine responsible for computing the *F* (or *K*) matrix, the remaining working memory space is divided into N_{thread} blocks. Each of these "Fock matrix blocks" will be used as private space for a single thread to store its contribution to the *F* (or *K*) matrix, and for the intermediate arrays used in *F* (or *K*) matrix construction.
3. Create N_{thread} threads using an OpenMP `parallel` directive. Each thread reads in the relevant parts of the shared density matrix block, calculates a subset of the ERIs and does a subset of the numerical integration, computing a contribution to its own local *F* (or *K*) matrix.
4. The child threads terminate when all their ERIs have been evaluated and their portion of the numerical integration is complete. The master thread then sums the N_{thread} private copies of the *F* (or *K*) matrices. This step is performed sequentially on master thread since the cost of adding a few $O(n^2)$ *F* (or *K*) matrices is usually small compared to the $O(n^4)$ cost of forming them.

In the case of a force evaluation the approach is similar, except each thread is now making contributions to the gradient vector rather than the F (or K) matrix.

We note that since the density matrix is read only, while the *F* (or K) matrices are private to each thread during the parallel section, in the CLOMP implementation once the initial penalty associated with fetching the relevant pages across the network to the right node has occurred, there should be little contention between threads. Thus HF and DFT energy and gradient calculations within *Gaussian* have the potential to achieve high parallel performance when using CLOMP provided that the time required to evaluate the ERIs and/or do the numerical integration is sufficiently large to mask the network overheads.

To use CLOMP in the code we firstly replace the **`malloc()`** system call by the CLOMP analogue **`kmp_sharable_malloc();`** this makes the whole of the *Gaussian* working array shared between all threads in the Link. A large amount of the shared variables were automatically tagged as sharable by the compiler, however, a significant number of other variables had to be identified by hand (around 60 sharable directives were inserted for Link 502). The parallel Links were then compiled and linked by using the CLOMP Intel 10.0 compiler with the **`-cluster-openmp,`** **`-clomp-sharable-commons,`** **`-clomp-sharable-localsaves`** and **`-clomp-sharable-argexprs`** directives.

For the purpose of this study, a number of other OpenMP directives were deliberately disabled since they correspond to fine grain loop parallelization that is known not to run well using CLOMP over multiple nodes in a cluster. In due course these directives should be reactivated, but with parallelism limited to just the number of threads within the master node.

3 Experimental Details

All performance experiments were carried out using a Linux cluster containing 4 nodes each with a 2.4GHz Intel Core2 Quad-core Q6600 CPU and 4GB of local DRAM memory. The cluster nodes were connected via Gigabit Ethernet.

The Intel C/Fortran compiler 10.0 was used to compile and build the two ported parallel Links of the *Gaussian* development version (GDV) with the CLOMP flags given above.

Five different HF and DFT benchmark jobs were considered spanning a typical range of molecular system. These are detailed in Table 1.

Table 1. Benchmarks used for performance measurements of the CLOMP implementation of Gaussian program

Case	Method	Basis	Molecule	Links	Routines Used
I	HF	6-311g*	Valinomycin	502	PRISM
II	BLYP	6-311g*	Valinomycin	502	PRISMC, CALDFT
III	BLYP	cc-pvdz	C60	502&703	PRISMC, CALDFT
IV	B3LYP	3-21g*	Valinomycin	502&703	PRISM, CALDFT
V	B3LYP	6-311g**	α-pinene	703	PRISM, CALDFT

Only the first SCF iteration is measured within Link 502 (the time to complete a full SCF calculation will scale almost linearly with the number of iterations required for convergence). The reported times comprises both the parallel formation of the F (or K) matrix and its (sequential) diagonalization.

4 Results and Discussion

Table 2 shows the speedups, defined as the ratio of the serial run time (t_s) divided by the parallel run time (t_p) for the five benchmark tests and Links 502 and 703 as applicable. The effect of using multiple cores and multiple nodes is contrasted. We use the single-thread non-CLOMP OpenMP executing time to calculate the speedup in the present work. We note also that the *Gaussian* code uses cache blocking when evaluating integrals in order to maximize performance [6]. For the quad-core processor used in this work the presence of a shared level 2 cache means that the optimal cache blocking size varies with the number of cores being used. In an attempt to remove this effect, results have been calculated by using the optimal cache blocking size for a given number of threads on a node.

Table 2. Speedups of Link 502(L502) and Link 703(L703) for benchmark systems

N_{node}×N_{core}		Case *I*	Case *II*	Case *III*		Case *IV*		Case *V*
		L502	**L502**	**L502**	**L703**	**L502**	**L703**	**L703**
1-thread	1×1	0.93	0.77	0.79	1.02	0.86	0.98	0.93
2-thread	1×2	1.73	1.48	1.55	1.99	1.61	1.90	1.75
	2×1	1.78	1.56	1.56	1.97	1.45	1.92	1.79
4-thread	1×4	2.91	2.53	2.83	3.26	2.55	3.07	2.67
	2×2	3.16	2.83	2.93	3.88	2.30	3.67	3.28
	4×1	3.25	2.94	2.98	3.91	2.21	3.71	3.12
8-thread	2×4	4.83	4.26	4.95	6.41	2.87	5.83	4.49
	4×2	5.55	4.75	5.24	7.68	3.13	7.09	5.30
16-thread	4×4	7.30	5.25	7.71	12.47	2.88	10.76	6.74

The results for the 1-thread case given in Table 2 show that in some cases there exists a significant slowdown associated with using CLOMP over “normal” OpenMP. Similar results have been reported elsewhere [7]. For all multi-thread cases speedups are observed. For two threads this varies from 1.45 to 1.99. The difference between using two cores within one node versus one core each on two nodes is surprisingly small, with the biggest difference being for case IV where using two cores within one node is preferable.

With 4 threads speedups of between 2.21 and 3.91 are observed. In most cases performance is better when using multiple nodes compared to using all cores within one node. Similar results are also found with 8 threads. This suggests that either memory bandwidth and/or contention due to the shared cache is limiting scalability when using 4 cores on a single chip.

For 16 threads the best speedup observed is 12.47 for Link 703 of case III. However we also see a relatively poor speedup of just 2.88 for Link 502 of Case IV. This is due to the fact that for this system Link 502 takes relatively little time (~100s), and the overhead of parallelization with 16 cores is large. Also, for this calculation the sequential time becomes significant.

Overall the results indicate that for large HF and DFT *Gaussian* energy and gradient calculations CLOMP is able to provide reasonable parallel performance over four nodes. To analysis the results further we use the performance model of Cai el al. [4, 8]. In this model execution time is based on the number, types and the associated approximate costs of page faults that occur during the execution of each parallel section. For *Gaussian* an outline of the important features of the two parallel Links and the sources of the page faults is given in Fig. 1.

```
Start of the Link:
     Allocate working array in the sharable heap using
   kmp_sharable_malloc();
Within the Link:
…
    Obtain new density matrix
    Parallel loop over $N_{thread}$ (OpenMP Parallel region):
      Call Prism, PrismC or CalDFT to calculate 1/$N_{thread}$ of the
      total integrals and save their contributions to each
      thread's private Fock matrix;
    End Parallel loop
    loop over i=2, $N_{thread}$ (sequential region)
       Add Fock matrix created by thread i to the master
      thread's Fock matrix;
    endloop
...
Exit Link;
```

Fig. 1. Basic CLOMP parallel construct in Link 502 of the Gaussian program (Link703 is similar, except contributions are now to the gradient vector)

At the beginning of each Link the master threads obtains a new copy of the density matrix. This will cause the underlying DSM to pass write notices to all other threads at the implicit barrier that occurs at the top of the next parallel region indicating that the corresponding memory pages have been modified. Within the parallel region each thread will read elements of the density matrix and update elements of their *F* (or *K*) matrix. This will cause a series of page faults to occur as new memory pages are accessed. Over time each thread is likely to access all density matrix elements and make contributions to all *F* (or *K*) matrix elements. For the density matrix this means each thread will end up fetching over its own read-only copy of the entire density matrix from the master thread. While for the *F* (or *K*) matrix each thread will end up creating its own writeable copy of this matrix. After the CLOMP parallel region has finished, a moderate amount of fetch page faults will occur on the master thread as it seeks to sum together the partial contributions to the *F* (or *K*) matrices computed by the other threads.

In the analysis model of Cai et al [4], the overhead associated with a CLOMP parallel calculation is determined by the thread that encounters the largest number of page faults (the critical path). The total CLOMP execution time on p threads is given by:

$$Tot(p) = \frac{Tot(1)^{par}}{p} + T^{crit}(p) = \frac{Tot(1)^{par}}{p} + Max_{i=0}^{p-1}(N_i^w C^w + N_i^f C^f) \qquad (5)$$

where $Tot(1)^{par}$ is the sum of the elapsed times for the parallel regions when the application is run using just one thread, p is the number of threads actually used, $T^{crit}(p)$ is the page fault time cost for the thread that encounters the maximum number of page faults in the p-thread calculations. $T^{crit}(p)$ is further expanded in terms of N_i^w and N_i^f, the number of write and fetch page faults respectively for thread *i* in the parallel region and their corresponding costs, C^w and C^f. The $Max_{i=0}^{p-1}$ in Eqn. (5) follows the thread in each parallel region that has the maximum page fault cost.

To obtain the page fault number for the five *Gaussian* test cases we have used the SEGVprof profiling tool that is distributed as part of the CLOMP distribution. This tool creates profile files (.gmon) for all CLOMP processes, reporting the segmentation faults occurring for each thread in each parallel region. SEGVprof also provides a script (**`segvprof.pl`**) that reports aggregated page fault counts. This script was extended to produce per-thread results. The values of C^w and C^f are system dependent and have been measured using two-nodes with the code given in Ref. [8]. This gave the values of C^w=10μs and C^f=171μs. These values are expected to be lower limits on the cost of page faults.

Since the model of Cai et al. [4] assumes there is no sequential time we give in Table 3 execution times and associated critical path page fault counts for just the PRISM, PRISMC and CALDFT portions of Links 502 and 703. This shows that the elapsed times cover a large range from just over 6 seconds to nearly 2900 seconds. Not surprisingly total page fault counts roughly reflect the size of the system, which in return is related to execution time. For any given test case and routine, the number of write and fetch page faults appears to remain roughly constant when going from 2 to 4 threads (1 thread/node). This is in line with the expectation expressed above, i.e. that

Table 3. The elapsed time (*Tot*(n)) and measured page fault numbers of subroutine PRISM, PRISMC and CALDFT in Link 502 (L502) and Link 703(L703) using 2-thread and 4-thread (1 thread/node) for all experimental cases. Also shown are ΔTot and ΔT^{crit} defined in Eqn. (6) (see text for further details).

Experiment (Links)	Routines	N_{thread}	*Tot*(n) (sec.)	Max page faults per thread		From Eqn. 6 (sec.)	
				Write	Fetch	ΔTot	ΔT^{crit}
Case I (L 502)	PRISM	2 4	751.8 386.3	4931 4934	13148 13150	20.8	2.30
Case II (L 502)	PRISMC	2 4	285.2 146.3	5104 5347	6036 6273	7.4	1.17
	CALDFT	2 4	185.1 93.9	8043 7540	3933 3680	2.7	0.66
Case III (L 502)	PRISMC	2 4	286.1 147.5	5337 5628	5240 5253	8.9	0.96
	CALDFT	2 4	95.0 49.5	3852 4146	1493 2187	4.0	0.54
Case III (L 703)	PRISMC	2 4	2894.3 1454.0	6038 6043	8577 8580	13.7	1.53
	CALDFT	2 4	272.8 137.0	6682 6475	1011 1127	1.2	0.28
Case IV (L 502)	PRISM	2 4	53.9 29.0	1497 1500	3576 3579	4.1	0.63
	CALDFT	2 4	86.1 43.3	3170 3069	916 1007	0.5	0.22
Case IV (L 703)	PRISM	2 4	353.9 181.9	1853 1854	5224 5229	9.9	0.91
	CALDFT	2 4	109.6 54.2	8497 8954	1077 1167	-1.2	0.31
Case V (L 703)	PRISM	2 4	21.2 11.1	1151 1151	548 551	1.0	0.11
	CALDFT	2 4	12.0 6.3	3849 3344	91 97	0.6	0.05

each thread will read most density matrix elements and produce contributions to most *F* (or *K*) matrix elements. In general the number of fetch faults are more than the number of write faults for PRISM and PRISMC, but much less for CALDFT. This reflects the fact that PRISM or PRISMC is called before CALDFT, so the fetch faults associated with creating a read only copy of the density matrix occur during execution of PRISM/PRISMC not CALDFT. (It is also interesting that in comparison to the page fault counts given by Cai et al. [8] for the NAS parallel benchmark codes the absolute value of the counts for *Gaussian* are much smaller, despite the fact that the execution time is comparable.)

To investigate the applicability of the Cai et al. [4] model we note that there should exist the following equality between 2-thread and 4-thread calculations:

$$\Delta Tot = 2\times Tot(4) - Tot(2) = 2\times T^{crit}(4) - T^{crit}(2) = \Delta T^{crit} \qquad (6)$$

Thus for the model of Cai et al. [4] to hold the scaled difference between the measured elapsed times reported in Table 2 when using 4 and 2 threads should be equal to the time difference computed using write and fetch page fault numbers also given in Table 2, combined with the cost penalty values of C^w=10μs and C^f=171μs. To test this we plot in Fig. 2 these two values. The results clearly show a large difference, although interestingly there does appear to be a linear relationship between the two quantities with a scale factor of around 8.4.

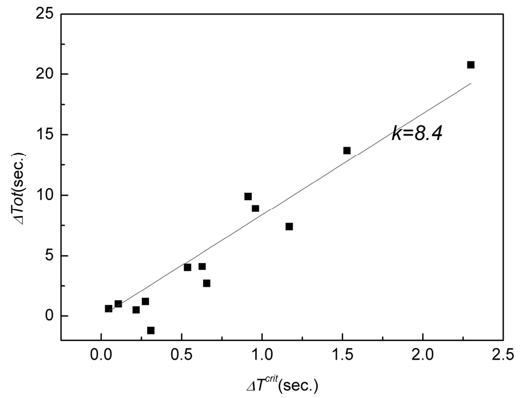


Fig. 2. ΔTot ~ ΔT^{crit} for the data given in Table 2

Possible causes for the inability of the Cai et al. [4] model to describe correctly the performance of *Gaussian* are its failure to account for load imbalance and repeated computation (reflected by the term $\frac{Tot(1)^{par}}{p}$ in Eqn. (5)), the assumption that page faults occurring on different threads are fully overlapped, and assignment of a fixed cost to servicing a fetch fault regardless the number of threads involved. In this case it is most likely that load imbalance and repeated computation are the main problems, since the total number of page faults appears to be too small compared to the total execution time for this to be a major cause for error. Specifically for case I and Link 502 ΔT^{cri} as evaluated using the page fault counts in Table 3 with costs of C^{w} =10μs and C^{f} =171μs is just 2.25s, yet ΔTot is 20.8 seconds.

5 Conclusions

We have investigated the performance of the CLOMP implemented quantum chemistry software package *Gaussian* on the 4-node Linux distributed memory system equipped with Intel Quad-core processors. Comparable or better scalability was found for using multi-nodes compared to multi-cores. Page fault measurements reveals relatively low counts within parallel regions, implying that HF and DFT energy and gradient computations within *Gaussian* are well suited to implementation with CLOMP as the effort associated with keeping the sharable memory consistency is low. A critical path model has been used to analyze performance, but the accuracy of this model appears to be limited due to load imbalance. Work is currently underway to extend the model of Cai et al. [4] to include load balancing, and also to study use of CLOMP for other parts of the *Gaussian* code.

Acknowledgments. This work is funded by Australian Research Council Linkage Grants LP0669726 and LP0774896 with support from Intel Corporation, *Gaussian* Inc. and Sun Microsystems.

References

1. Hoeflinger, J.P.: Extending openmp to clusters. White Paper, Intel. Corporation (2006)
2. Hoeflinger, J.P., Meadows, L.: Programming OpenMP on Clusters. HPCwire 15(20) (May 19, 2006), `http://www.hpcwire.com/hpc/658711.html`
3. Frisch, M.J., et al.: Gaussian 03. Gaussian, Inc., Wallingford CT (2004)
4. Cai, J., Rendell, A.P., Strazdins, P.E., Wong, H.J.: Performance model for cluster-enabled OpenMP implementations. In: Proceeding of 13th IEEE Asia-Pacific Computer Systems Architecture Conference, pp. 1–8 (2008)
5. Sosa, C.P., Andersson, S.: Some Practical Suggestions for Performing Gaussian Benchmarks on a pSeries 690 System, IBM Form Number REDP0424, April 24 (2002)
6. Yang, R., Antony, J., Janes, P.P., Rendell, A.P.: Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In: The 2008 International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 31–36. IEEE Computer Society, Los Alamitos (2008)
7. Terboven, C., an Mey, D., Schmidl, D., Wagner, M.: First Experiences with Intel. Cluster OpenMP. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 48–59. Springer, Heidelberg (2008)
8. Cai, J., Rendell, A.P., Strazdins, P.E., Wong, H.J.: Predicting performance of Intel Cluster OpenMP with Code Analysis method, ANU Computer Science Technical Reports, TR-CS-08-03 (2008)

Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs

Stephen L. Olivier and Jan F. Prins

University of North Carolina at Chapel Hill, Chapel Hill NC 27599, USA
{olivier,prins}@unc.edu

Abstract. The UTS benchmark is used to evaluate task parallelism in OpenMP 3.0 as implemented in a number of recently released compilers and run-time systems. UTS performs parallel search of an irregular and unpredictable search space, as arises e.g. in combinatorial optimization problems. As such UTS presents a highly unbalanced task graph that challenges scheduling, load balancing, termination detection, and task coarsening strategies. Scalability and overheads are compared for OpenMP 3.0, Cilk, and an OpenMP implementation of the benchmark without tasks that performs all scheduling, load balancing, and termination detection explicitly. Current OpenMP 3.0 implementations generally exhibit poor behavior on the UTS benchmark.

1 Introduction

The recent addition of task parallelism support to OpenMP 3.0 [1] offers improved means for application programmers to achieve performance and productivity on shared memory platforms such as multi-core processors. However, efficient execution of task parallelism requires support from compilers and run time systems. Design decisions for those systems include choosing strategies for task scheduling and load-balancing, as well as minimizing overhead costs.

Evaluating the efficiency of run time systems is difficult; the applications they support vary widely. Among the most challenging are those based on unpredictable and irregular computation. The Unbalanced Tree Search (UTS) benchmark [2] represents a class of such applications requiring continuous load balance to achieve parallel speedup. In this paper, we compare the performance and scalability of the UTS benchmark on three different OpenMP 3.0 implementations (Intel icc 11, Mercurium 1.2.1, SunStudio 12) and an experimental prerelease of gcc 4.4 that includes OpenMP 3.0 support. For comparison we also examine the performance of the UTS benchmark using Cilk [3] tasks and using an OpenMP implementation without tasks that performs all scheduling, load balancing, and termination detection explicitly. Throughout this paper we will refer to the latter as the thread-level parallel implementation. Additional experiments focus on comparing overhead costs. The primary contribution of the paper is an analysis of the experimental results for a set of compilers that support task parallelism.

The remainder of the paper is organized as follows: Section 2 outlines background and related work on run time support for task parallelism. Section 3 describes the UTS benchmark. Section 4 presents the experimental results and analysis. We conclude in Section 5 with some recommendations based on our findings.

2 Background and Related Work

Many theoretical and practical issues of task parallel languages and their run time implementations were explored during the development of earlier task parallel programming models, such as Cilk [4,3]. The issues can be viewed in the framework of the dynamically unfolding task graph in which nodes represent tasks and edges represent completion dependencies.

The *scheduling strategy* determines which ready tasks to execute next on available processing resources. The *load balancing strategy* keeps all processors supplied with work throughout execution. Scheduling is typically decentralized to minimize contention and locking costs that limit scalability of centralized schedulers. However decentralized scheduling increases the complexity of load balancing when a local scheduler runs out of tasks, determining readiness of tasks, and determining global completion of all tasks.

To decrease overheads, various *coarsening strategies* are followed to aggregate multiple tasks together, or to execute serial versions of tasks that elide synchronization support when not needed. However such coarsening may have negative impact on load balancing and availability of parallel work.

Cilk scheduling uses a *work-first* scheduling strategy coupled with a randomized work stealing load balancing strategy shown to be optimal[5]. A lazy task creation approach, developed for parallel implementations of functional languages [6], makes parallel slack accessible while avoiding overhead costs until more parallelism is actually needed. The compiler creates a fast and a slow clone for each task in a Cilk program. Local execution always begins via execution of the fast clone, which replaces task creation with procedure invocation. An idle processor may steal a suspended parent invocation from the execution stack, converting it to the slow clone for parallel execution.

In OpenMP task support, “cutoff” methods to limit overheads were proposed in [7]. When cutoff thresholds are exceeded, new tasks are serialized. One proposed cutoff method, *max-level*, is based on the number of ancestors, i.e. the level of recursion for divide-and-conquer programs. Another is based on the number of tasks in the system, specified as some factor k times the number of threads. The study in [7] finds that performance is often poor when no cutoff is used and that different cutoff strategies are best for different applications. Adaptive Task Cutoff (ATC) is a scheme to select the cutoff at runtime based on profiling data collected early in the program's execution [8]. In experiments, performance with ATC is similar to performance with manually specified optimal cutoffs. However, both leave room for improvement on unbalanced task graphs.

Iterative chunking coarsens the granularity of tasks generated in loops [9]. Aggregation is implemented through compiler transformations. Experiments show mixed results, as some improvements are in the noise compared to overheads of the run time system.

Intel's "workqueuing" model was a proprietary OpenMP extension for task parallelism [10]. In addition to the task construct, a taskq construct defined queues of tasks explicitly. A noteworthy feature was support for reductions among tasks in a task queue. Early evaluations of OpenMP tasking made comparisons to Intel workqueuing, showing similar performance on a suite of seven applications [11].

An extension of the Nanos Mercurium research compiler and run time [11] has served as the prototype compiler and run time for OpenMP task support. An evaluation of scheduling strategies for tasks using Nanos is presented in [7]. That study concluded that in situations where each task is *tied*, i.e. fixed to the thread on which it first executes, breadth-first schedulers perform best. They found that programs using *untied* tasks, i.e. tasks allowed to migrate between threads when resuming after suspension, perform better using work-first schedulers. A task should be tied if it requires that successive accesses to a threadprivate variable be to the same thread's copy of that variable. Otherwise, untied tasks may be used for greater scheduling flexibility.

Several production compilers have now incorporated OpenMP task support. IBM's implementation for their Power XLC compilers is presented in [12]. The upcoming version 4.4 release of the GNU compilers [13] will include the first production open-source implementation of OpenMP tasks. Commercial compilers are typically closed source, underscoring the need for challenging benchmarks for black-box evaluation.

3 The UTS Benchmark

The UTS problem [2] is to count the nodes in an implicitly defined tree: any subtree in the tree can be generated completely from the description of its parent. The number of children of a node is a function of the node's description; in our current study a node can only have zero or $m = 8$ children. The description of each child is obtained by an evaluation of the SHA-1 cryptographic hash function [14] on the parent description and the child index. In this fashion, the UTS search trees are implicitly generated in the search process but nodes need not be retained throughout the search.

Load balancing of UTS is particularly challenging since the distribution of subtree sizes follows a power law. While the variance in subtree sizes is enormous, the expected subtree size is identical at all nodes in the tree, so there is no advantage to be gained by stealing one node over another. For the purpose of evaluating run time load-balancing support, the UTS trees are a particularly challenging adversary.

3.1 Task Parallel Implementation

To implement UTS using task parallelism, we let the exploration of each node be a task, allowing the underlying run time system to perform load balancing as needed. A sketch of the implementation follows below:

```
void Generate_and_Traverse(Node* parentNode, int childNumber) {
  Node* currentNode = generateID(parentNode, childNumber);
  nodeCount++;   // threadprivate, combined at termination
  int numChildren = m with prob q, 0 with prob 1-q
  for (i = 0; i < numChildren; i++) {
    #pragma omp task untied firstprivate(i)
      Generate_and_Traverse(currentNode, i);
  }
}
```

Execution is started by creating a parallel region (with a threadprivate counter for the number of nodes counted by the thread). Within the parallel region a single thread creates tasks to count the subtrees below the root. A single taskwait is used to end the parallel region when the entire tree has been explored.

3.2 Thread-Level Parallel Implementation without Tasks

Unlike the task parallel implementation of UTS, the thread-level parallel implementation described in [2] using OpenMP 2.0 explicitly specifies choices for the order of traversal (depth-first), load balancing technique (work stealing), aggregation of work, and termination detection. A sketch of the implementation follows below:

```
void Generate_and_Traverse(nodeStack* stack) {
  #pragma omp parallel
    while (1) {
      if (empty(stack)) {
        ... steal work from other threads or terminate ...
      }
      currentNode = pop(stack);
      nodeCount++;   // threadprivate, gathered using critical later
      int numChildren = m with prob q, 0 with prob q-1
      for (i = 0; i < numChildren; i++) {
         ...initialize childNode...
         childNode = generateID(currentNode, i);
         push(stack, childNode);
      }
    }
}
```

Execution is started with the root node on the nodeStack of one thread; all other threads start with an empty stack. Note that the single parallel region manages load balancing among threads, termination detection, and the actual tree traversal.

3.3 Cilk Implementation

For comparison, we created a cilk implementation of UTS which is close to the OpenMP 3.0 task implementation. It differs mainly in its use of a Cilk inlet in the search function to accumulate partial results for the tree node count as spawned functions return. The Cilk runtime handles the required synchronization to update the count.

4 Experimental Evaluation

We evaluate OpenMP task support by running UTS and related experiments on an Opteron SMP system. The Opteron system consists of eight dual-core AMD Opteron 8220 processors running at 2.8 Ghz, with 1MB cache per core.

We installed gcc 4.3.2, the Intel icc 11.0 compiler, SunStudio 12 with Ceres C 5.10, and an experimental prerelease of gcc 4.4 (12/19/2008 build). We also installed the Mecurium 1.2.1 research compiler with Nanos 4.1.2 run time. Since Nanos does not yet natively support the x86-64 architecture, we built and used the compiler for 32-bit IA32. We used cilk 5.4.6 for comparison with the OpenMP implementations on both machines; it uses the gcc compiler as a back end. The -O3 option is always used. Unless otherwise noted, reported results represent the average of 10 trials.

For a few results in Section 4.4 of the paper, we used an SGI Altix running Intel icc 11 and Nanos/Mercurium built for IA64. Details for that system are presented in that section.

4.1 Sequential and Parallel Performance on UTS

Table 1 shows sequential performance for UTS on the Opteron SMP system; the execution rate represents the number of tree nodes explored per unit time. We use tree *T3* from [2], a 4.1 million node tree with extreme imbalance. This tree is used in experiments throughout the paper. The table gives results for both the task parallel and the thread-level parallel implementations. They were compiled with OpenMP support disabled.

Figure 1 shows the speedup gained on the task parallel implementation using OpenMP 3.0, as measured against the sequential performance data given in table 1. We observe no speedup from Sun Studio and gcc. Cilk outperforms the Intel OpenMP task implementation, but neither achieve more than 10X speedup, though both show improved speedup as up to 16 cores are used. Figure 2 shows the speedup, over 15X in most cases, using the thread-level parallel implementation.

Table 1. Sequential performance on Opteron (Millions of tree nodes per second)

Implementation	gcc 4.3.2	Intel icc 11.0	Sun Ceres 5.10	gcc 4.4.0
Task Parallel	2.60	2.45	2.38	2.60
Thread-Level Parallel	2.48	2.49	2.17	2.39

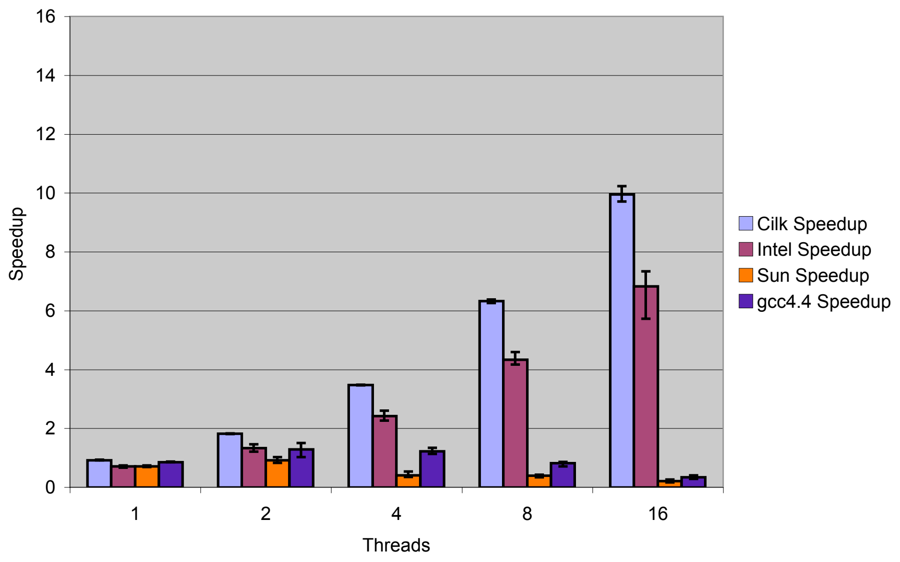


Fig. 1. UTS using cilk and several OpenMP 3.0 task implementations: Speedup on 16-way Opteron SMP. See Figure 7 and Section 4.4 for results using Nanos.

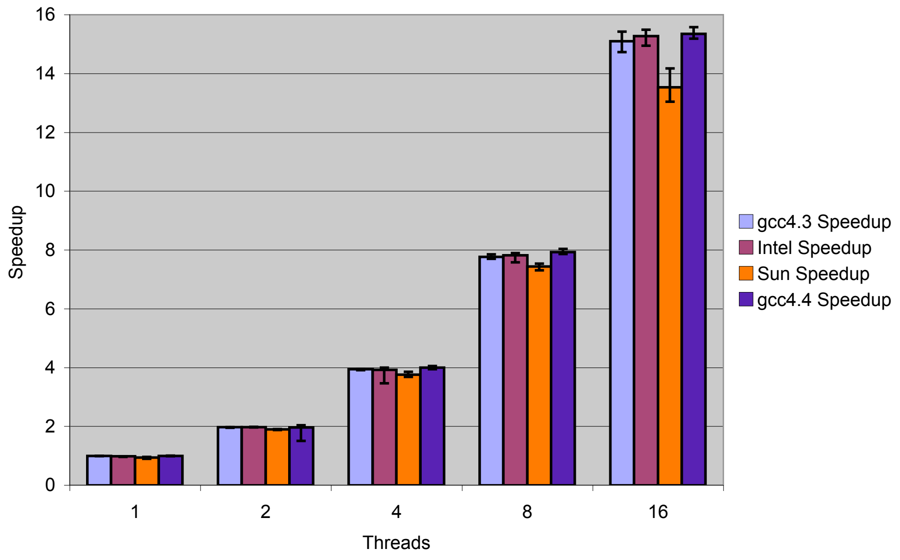


Fig. 2. UTS using thread-level OpenMP parallel implementation without tasks: Speedup on 16-way Opteron SMP. Work stealing granularity is a user-supplied parameter. The optimal value (64 tree nodes transferred per steal operation) was determined by manual tuning and used in these experiments.

4.2 Analysis of Performance

Two factors leading to poor performance are overhead costs and load imbalance. There is a fundamental tradeoff between them, since load balancing operations incur overhead costs. Though all run time implementations are forced to deal with that tradeoff, clever ones minimize both to the extent possible. Poor implementations show both crippling overheads and poor load balance.

Overhead Costs. Even when only a single thread is used, there are some overhead costs incurred using OpenMP. For task parallel implementation of UTS, Cilk achieves 96% efficiency, but efficiency is just above 70% using OpenMP 3.0 on a single processor using the Intel and Sun compilers and 85% using gcc 4.4. The thread-level parallel implementation achieves 97-99% on the Intel and gcc compilers and 94% on the Sun compiler.

To quantify the scaling of overhead costs in the OpenMP task run times, we instrumented UTS to record the amount of time spent on work (calculating SHA-1 hashes). To minimize perturbation from the timing calls, we increased the amount of computation by performing 100 SHA-1 hash evaluations of each node. Figure 3 presents the percent of total time spent on overheads (time not spent on SHA-1 calculations). Overhead costs grow sharply in the gcc implementation, dwarfing the time spent on work. The Sun implementation also suffers from high overheads, reaching over 20% of the total run time. Overheads grow slowly from 2% to 4% in the Intel run time. Note that we increased the granularity of computation 100-fold, so overheads on the original fine-grained problem may be much higher still.

Load Imbalance. Now we consider the critical issue of load imbalance. To investigate the number of load balancing operations, we modified UTS to record

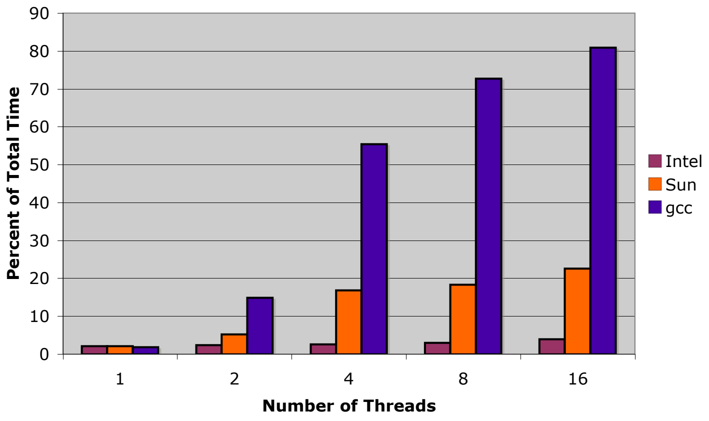


Fig. 3. Overheads (time not calculating SHA-1 hash) on UTS using 100 repetitions of the SHA-1 hash per tree node

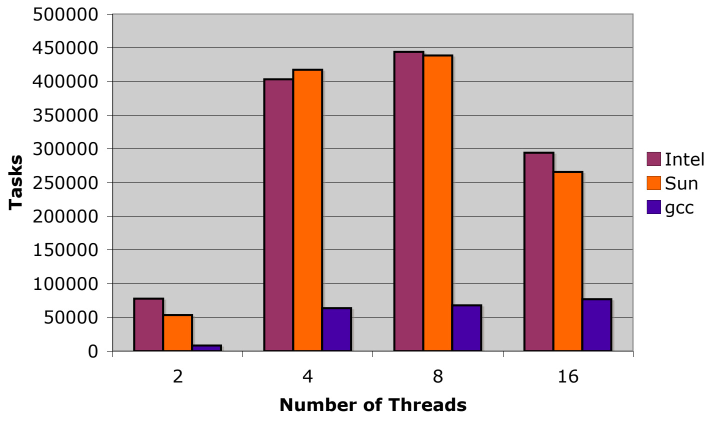


Fig. 4. Number of tasks started on different threads from their parent tasks or migrating during task execution, indicating load balancing efforts. 4.1M tasks are performed in each program execution.

the number of tasks that start on a different thread than the thread they were generated from or that migrate when suspended and subsequently resumed. Figure 4 shows the results of our experiments using the same 4.1M node tree (T3), indicating nearly 450k load balancing operations performed by the Intel and Sun run times per trial using 8 threads. That comprises 11% of the 4.1M tasks generated. In contrast, gcc only performs 68k load balancing operations. For all implementations, only 30%-40% of load balancing operations occur before initial execution of the task, and the rest as a result of migrations enabled by the *untied* keyword.

Given the substantial amount of load balancing operations performed, we investigated whether they are actually successful in eliminating load imbalance. To that end, we recorded the number of nodes explored at each thread, shown in Figure 11. Note that since ten trials were performed at each thread count, there are 10 data points shown for trials on one thread, 20 shown for trials on two threads, etc. The results for the Intel implementation (a) show good load balance, as roughly the same number of nodes (4.1M divided by the number of threads) are explored on each thread. With the Sun implementation, load balance is poor and worsens as more threads are used. Imbalance is poorer still with gcc.

Even if overhead costs were zero, speedup would be limited by load imbalance. The total running time of the program is at least the work time of the thread that does the most work. Since each task in UTS performs the same amount of work, one SHA-1 hash operation, we can easily determine that efficiency e is limited to the ratio of average work per thread to maximum work per thread. The lost efficiency $(1-e)$ for the different OpenMP task implementations is shown in Figure 5. Poor load balance by the Sun and gcc implementations severely limits scalability. Consider the 16-thread case: neither implementation

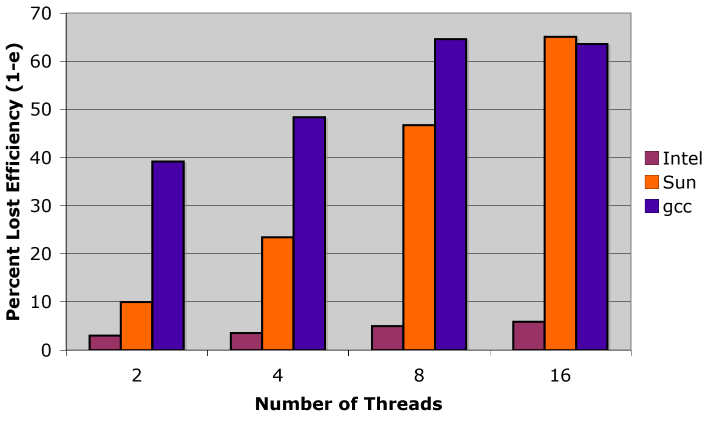


Fig. 5. UTS on Opteron: Lost efficiency due to load imbalance

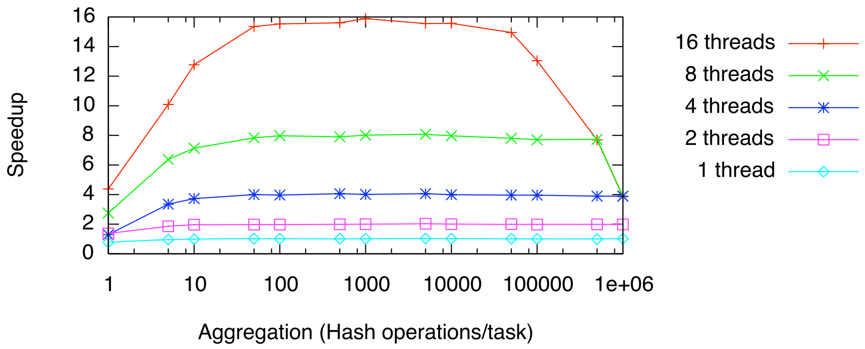


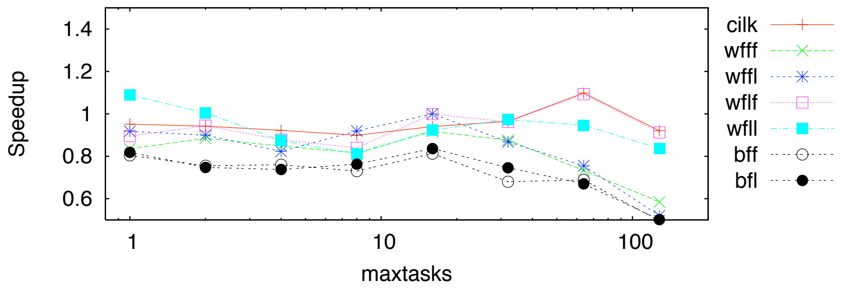
Fig. 6. Work aggregated into tasks. Speedup on Opteron SMP using the Intel OpenMP tasks implementation. Results are similar using the gcc 4.4 and Sun compilers, though slightly poorer at the lower aggregation levels.

can achieve greater than 40% efficiency even if overhead costs were nonexistent. On the other hand, inefficiency in the Intel implementation cannot be blamed on load imbalance.

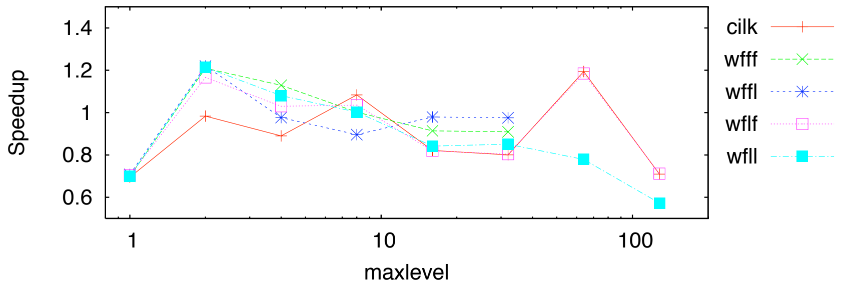
4.3 Potential for Aggregation

The thread parallel implementation reduces overhead costs chiefly by aggregating work. Threads do not steal nodes one at at time, but rather in chunks whose size is specified as a parameter. A similar method could be applied within an OpenMP run time, allowing chunks of tasks to be moved between threads at a time.

To test possible overhead reduction from aggregation, we designed an experiment in which 4M SHA-1 hashes are performed independently. To parallelize we use a loop nest in which the outer forall generates tasks. Each task executes a



(a) maxtasks cutoff



(b) maxlevel cutoff

Fig. 7. UTS speedup on Opteron SMP using two threads with different scheduling and cutoff strategies in Nanos. Note that "cilk" denotes the cilk-style scheduling option in Nanos, not the cilk compiler.

Table 2. Nanos scheduling strategies. For more details see [7].

Name	Description
wfff	work-first with FIFO local queue access, FIFO remote queue access
wffl	work-first with FIFO local queue access, LIFO remote queue access
wflf	work-first with LIFO local queue access, FIFO remote queue access
wfll	work-first with LIFO local queue access, LIFO remote queue access
cilk	wflf with priority to steal parent of current task
bff	breadth-first with FIFO task pool access
bfl	breadth-first with LIFO task pool access

loop of k SHA-1 hashes. So k represents an aggregation factor. Since the outer forall has 4M / k iterations equally distributed by static scheduling, there should be little or no load balancing. Thus, performance measurements should represent a lower bound on the size of k needed to overcome overhead costs. Figure 6 shows speedup for aggregation of $k = 1$ to 100000 run using the Intel implementation. (Results for the gcc 4.4 and Sun compilers are very similar and omitted for lack of space.) Speedup reaches a plateau at $k = 50$. We could conclude that for our

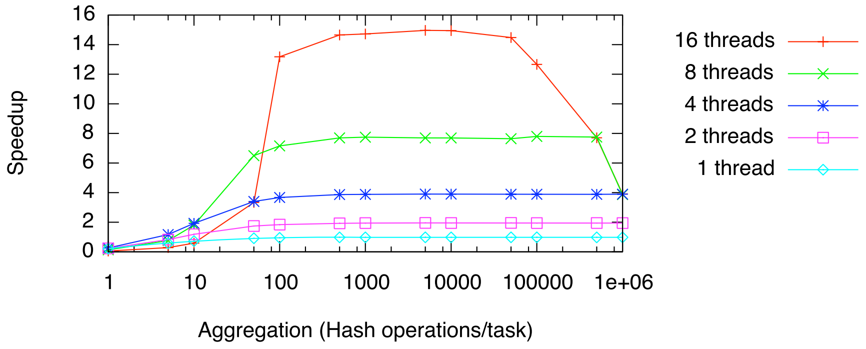


Fig. 8. Work aggregated into tasks. Speedup on Opteron SMP using cilk-style scheduling in Nanos. Results are similar using other work-first scheduling strategies.

tree search, enough tasks should be moved at each load balancing operation to yield 50 tree nodes for exploration. Notice that for 8 and 16 threads, performance degrades when k is too high, showing that too much aggregation leads to load imbalance, i.e. when the total number of tasks is a small non-integral multiple of the number of threads.

4.4 Scheduling Strategies and Cutoffs

As mentioned in Section 2, the Mercurium compiler and Nanos run time offer a wide spectrum of runtime strategies for task parallelism. There are breadth-first schedulers with FIFO or LIFO access, and work-first schedulers with FIFO or LIFO local access and FIFO or LIFO remote access for stealing. There is also a "Cilk-like" work-first scheduler in which an idle remote thread attempts to steal the parent task of a currently running task. In addition, the option is provided to serialize tasks beyond a cutoff threshold, a set level of the task hierarchy (maxlevel) or a certain number of total tasks (maxtasks). Note that a maxtasks cutoff is imposed in the gcc 4.4 OpenMP 3.0 implementation, but the limit is generous at 64 times the number of threads.

Figure 7 shows the results of running UTS using two threads in Nanos with various scheduling strategies and varying values for the maxtasks and maxlevel cutoff strategies. See Table 2 for a description of the scheduling strategies represented. The breadth-first methods fail due to lack of memory when the maxlevel cutoff is used. There are 2000 tree nodes at the level just below the root, resulting in a high number of simultaneous tasks in the breadth-first regime. As shown in the graphs, we did not observe good speedup using Nanos regardless of the strategies used. Though not shown, experiments confirm no further speedup using four threads.

Limiting the number of tasks in the system (maxtasks cutoff) may not allow enough parallel slack for the continuous load balancing required. At the higher end of the range we tested in our experiments, there should be enough parallel

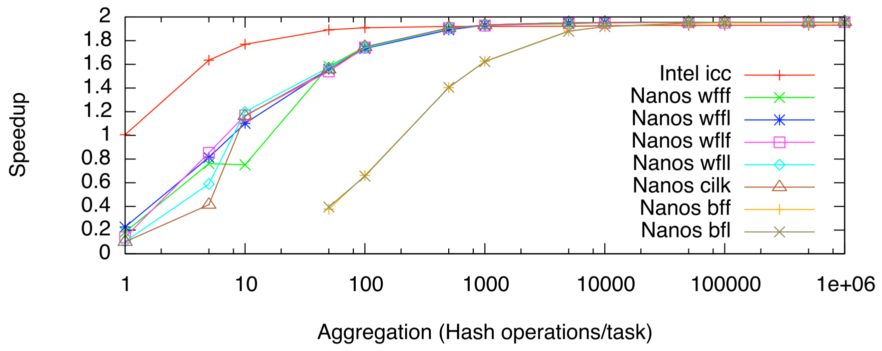


Fig. 9. Work aggregated into tasks. Speedup on an SGI Altix for 4M hash operations performed; work generated evenly upon two threads. The various Nanos scheduling strategies are used without cutoffs, and Intel icc is shown for comparison. Note that "cilk" denotes the cilk-style scheduling option in Nanos, not the cilk compiler.

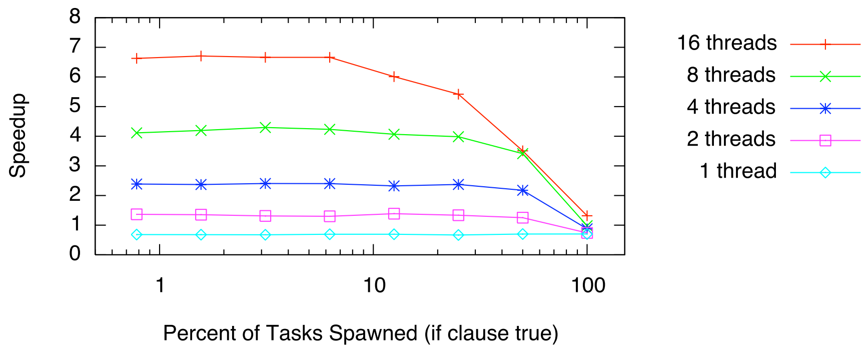
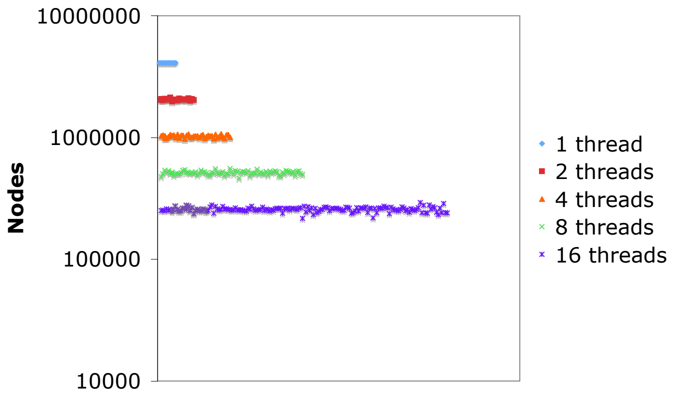


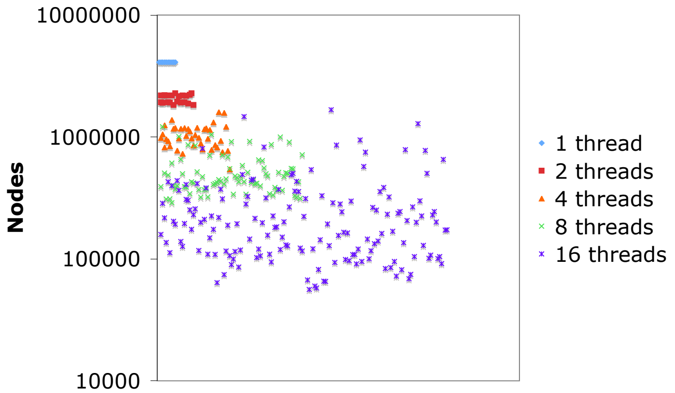
Fig. 10. UTS Speedup on Opteron SMP using the Intel OpenMP 3.0 task implementation with user-defined inlining specified using the *if()* clause

slack but overhead costs are dragging down performance. Cutting off a few levels below the root (maxlevel cutoff) leaves highly unbalanced work, since the vast majority of the nodes are deeper in the tree, and UTS trees are imbalanced everywhere. Such a cutoff is poorly suited to UTS. For T3, just a few percent of the nodes three levels below the root subtend over 95% of the tree. Adaptive Task Cutoff [8] would offer little improvement, since it uses profiling to predict good cutoff settings early in execution. UTS is unpredictable: the size of the subtree at each node is unknown before it is explored and variation in subtree size is extreme.

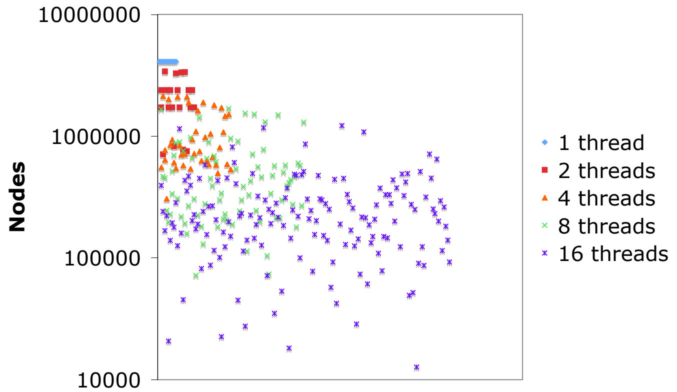
We also repeated aggregation experiments from Section 4.3 using Nanos. Figure 8 shows speedup using the cilk-like scheduling strategy with no cutoffs imposed. Results for other work-first schedulers is similar. Noticed that compared to the results from the same experiment using Intel compiler (Figure 6),



(a) Intel icc



(b) Sun



(c) gcc 4.4

Fig. 11. UTS on Opteron SMP: Number of nodes explored at each thread

speedup is much poorer at lower levels of aggregation with Nanos. Whereas speedup at 10 hash operations per second is about 13X with 16 threads using the Intel compiler, Nanos speedup is less than 1X.

Since the breadth-first methods struggle with memory constraints on the Opteron SMP, we tried the aggregation tests on another platform: an SGI Altix with lightweight thread support on the Nanos-supported 64-bit IA64 architecture. The Altix consists of 128 Intel Itanium2 processors running at 1.6 Ghz, each with 256kB of L2 cache and 6MB of L3 cache. We installed the Mecurium 1.2.1 research compiler with Nanos 4.1.2 run time and the Intel icc 11.0 compiler. Even using the breadth-first schedulers and no cutoffs, the tasks are able to complete without exhausting memory. Figure 9 shows experiments performed on two threads of the Altix. The Intel implementation outperforms Nanos at fine-grained aggregation levels. Among the Nanos scheduling options, the work-first methods are best.

4.5 The *if()* Clause

The OpenMP task model allows the programmer to specify conditions for task serialization using the *if()* clause. To evaluate its impact, we used the *if()* clause in a modified version of our task parallel implementation so that only $n\%$ percent of the tree nodes are explored in new tasks while the rest are explored in place. We varied n exponentially from less than 0.01% to 100%. Figure 10 shows the results on the Opteron SMP using the Intel compiler. Using the *if()* clause to limit the number of tasks seems to improve speedup. However, Figure 1 showed similar speedups using the same compiler and UTS implementation but with no *if()* clause. Why would setting $n = 100\%$ not yield the same results? We suspect that the use of the *if()* clause may disable a default internal cutoff mechanism in the Intel run time system.

5 Conclusions

Explicit task parallelism provided in OpenMP 3.0 enables easier expression of unbalanced applications. Consider the simplicity and clarity of the task parallel UTS implementation. However, there is clearly room for further improvement in performance for applications with challenging demands such as UTS.

Our experiments suggest that efficient OpenMP 3.0 run time support for very unbalanced task graphs remains an open problem. Among the implementations tested, only the Intel compiler shows good load balancing. Its overheads are also lower than other implementations, but still not low enough to yield ideal speedup. Cilk outperforms all OpenMP 3.0 implementations; design decisions made in its development should be examined closely when building the next generation of OpenMP task run time systems. A key feature of Cilk is its on-demand conversion of serial functions (fast clone) to concurrent (slow clone) execution. The "Cilk-style" scheduling option in Nanos follows the work stealing strategy of Cilk, but decides before task execution whether to inline a task or spawn it for concurrent execution.

We cannot be sure of the scheduling mechanisms used in the commercial OpenMP 3.0 implementations. The gcc 4.4 implementation uses a task queue and maintains several global data structures, including current and total task counts. Contention for these is a likely contributor to overheads seen in our experiments. Another problematic feature of the gcc OpenMP 3.0 implementation is its use of barrier wake operations upon new task creation to enable idle threads to return for more work. These operations are too frequent in an applications such as UTS that generates work irregularly. Even with an efficient barrier implementation, they may account for significant costs.

Experiments using several different scheduling strategies with cutoffs also show poor performance. Unbalanced problems such as UTS are not well suited to cutoffs because they make it difficult to keep enough parallel slack available. Aggregation of work should be considered for efficient load balancing with reduced overhead costs. Further work is needed to determine other ways in which OpenMP 3.0 run time systems could potentially be improved and whether additional information could be provided to enable better performance.

While the UTS benchmark is useful as a benchmarking and diagnostic tool for run time systems, many of the same problems it uncovers impact real world applications. Combinatorial optimization and enumeration lie at the heart of many problems in computational science and knowledge discovery. For example, protein design is a combinatorial optimization problem in which energy minimization is used to evaluate many combinations of amino acids arranged along the backbone to determine whether a desired protein geometry can be obtained [15]. An example of an enumeration problem in knowledge discovery is subspace clustering, in which subsets of objects that are similar on some subset of features are identified [16]. Another example is the Quadratic Assignment Problem (QAP) at the heart of transportation optimization. These sorts of problems typically require exhaustive search of a state space of possibilities. When the state space is very large, as is often the case, a parallel search may be the only hope for a timely answer.

Evaluation on a wider range of applications is needed to determine the shared impact of the compiler and run time issues that UTS has uncovered. One issue that we have not addressed in our experiments is locality. UTS models applications in which a task only requires a small amount data from its parent and no other external data. We anticipate future work in which we consider applications with more demanding data requirements.

Acknowledgements

Stephen Olivier is funded by a National Defense Science and Engineering Graduate Fellowship. The Opteron SMP used in this paper is a part of the BASS cluster purchased with funding from the National Institutes of Health's National Center for Research Resources, through their High-End Instrumentation program award number NIH 1S10RR023069-01 and administered by the UNC Department of Computer Science. The authors would like to thank the anonymous reviewers for their insightful comments, many of which have led to important improvements in the paper.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0 (May 2008)
2. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
3. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proc. 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI 1998), pp. 212–223 (1998)
4. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: PPoPP 1995: Proc. 5th ACM SIGPLAN symp. Princ. Pract. Par. Prog. (1995)
5. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Ann. Symp. Found. Comp. Sci., November 1994, pp. 356–368 (1994)
6. Mohr, E., Kranz, D.A., Robert, H., Halstead, J.: Lazy task creation: a technique for increasing the granularity of parallel programs. In: LFP 1990: Proc. 1990 ACM Conf. on LISP and Functional Prog., pp. 185–197. ACM, New York (1990)
7. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP task scheduling strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 100–110. Springer, Heidelberg (2008)
8. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conf. on Supercomputing, Piscataway, NJ, USA, pp. 1–11. IEEE Press, Los Alamitos (2008)
9. Ibanez, R.F.: Task chunking of iterative constructions in openmp 3.0. In: First Workshop on Execution Environments for Distributed Computing (July 2007)
10. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for Intel(R) SMP architectures. In: European Workshop on OpenMP, EWOMP 2002 (2002)
11. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: Adve, V.S., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 63–77. Springer, Heidelberg (2008)
12. Teruel, X., Unnikrishnan, P., Martorell, X., Ayguadé, E., Silvera, R., Zhang, G., Tiotto, E.: OpenMP tasks in IBM XL compilers. In: CASCON 2008: Proc. 2008 Conf. of Center for Adv. Studies on Collaborative Research, pp. 207–221. ACM, New York (2008)
13. Free Software Foundation, Inc.: GCC, the GNU compiler collection, `http://www.gnu.org/software/gcc/`
14. Eastlake, D., Jones, P.: US secure hash algorithm 1 (SHA-1). RFC 3174, Internet Engineering Task Force (September 2001)
15. Baker, D.: Proteins by design. The Scientist, 26–32 (July 2006)
16. Agrawal, R., Gehrke, J., Gunopulos, D., Raghavan, P.: Automatic subspace clustering of high dimensional data. Data Min. Knowl. Discov. 11(1), 5–33 (2005)

Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective

François Broquedis[1], Nathalie Furmento[3], Brice Goglin[2], Raymond Namyst[1], and Pierre-André Wacrenier[1]

[1] University of Bordeaux
[2] INRIA
[3] CNRS
LaBRI – 351 cours de la Libération
F-33405 Talence, France
`lastname@labri.fr`
`firstname.lastname@inria.fr`

Abstract. Exploiting the full computational power of current hierarchical multiprocessor machines requires a very careful distribution of threads and data among the underlying non-uniform architecture so as to avoid memory access penalties. Directive-based programming languages such as OpenMP provide programmers with an easy way to structure the parallelism of their application and to transmit this information to the runtime system.

Our runtime, which is based on a multi-level thread scheduler combined with a NUMA-aware memory manager, converts this information into "scheduling hints" to solve thread/memory affinity issues. It enables dynamic load distribution guided by application structure and hardware topology, thus helping to achieve performance portability. First experiments show that mixed solutions (migrating threads and data) outperform *next-touch*-based data distribution policies and open possibilities for new optimizations.

Keywords: OpenMP, Memory, NUMA, Hierarchical Thread Scheduling, Multi-Core.

1 Introduction

Modern computing architectures are increasingly parallel. While the *High Performance Computing* landscape is still dominated by large clusters, the degree of parallelism within cluster nodes is increasing. This trend is obviously driven by the emergence of multicore processors that dramatically increase the number of cores, at the expense of a poorer memory bandwidth per core. To minimize memory contention, hardware architects have been forced to go back to a hierarchical organization of cores and memory banks or, in other words, to NUMA architectures (*Non-Uniform Memory Access*). Note that such machines are now becoming mainstream thanks to the spreading of AMD HyperTransport and Intel QuickPath technologies.

Running parallel applications efficiently on older multiprocessor machines was essentially a matter of careful task scheduling. In this context, parallel runtime systems such as Cilk [7] or TBB [9] have proved to be very effective. In fact, these approaches can still behave well over hierarchical multicore machines in the case of cache-oblivious applications. However, in the general case, successfully running parallel applications on NUMA architectures requires a careful distribution of tasks and data to avoid "NUMA penalties". Moreover, applications with strong memory bandwidth requirements need data to be physically allocated on the "right" memory banks in order to reduce contention. This means that high-level information about the application behavior, in terms of memory access patterns or affinity between threads and data, must be conveyed to the underlying runtime system.

Several programming approaches provide means to specify task-memory affinities within parallel applications (OpenMP, HPF [10], UPC [3]). However, retrieving affinity relations at runtime is difficult; compilers and runtime systems must tightly cooperate to achieve a sound distribution of thread and data that can dynamically evolve according to the application behavior. Our prior work [17,2] emphasized the importance of establishing a persistent cooperation between an OpenMP compiler and the underlying runtime system on multicore NUMA machines. We designed ForestGOMP [17] that extends the GNU OpenMP implementation, GOMP, to make use of the BubbleSched flexible scheduling framework [18]. Our approach has proved to be relevant for applications dealing with nested, massive parallelism.

We introduce in this paper a major extension of our OpenMP runtime system that connects the thread scheduler to a NUMA-aware memory management subsystem. This new runtime not only can use per-bubble memory allocation information when performing thread re-distributions, but it can also perform data migration — either immediately or upon *next-touch*— in situations when it is more appropriate. We discuss several of these situations, and give insights about the most influential parameters that should be considered on today's hierarchical multicore machines. The remainder of this paper is organized as follows. We present the background of our work in Section 2. We describe our extensions to the ForestGOMP runtime system in Section 3. In Section 4, we evaluate the relevance of our proposal with several performance-oriented experiments. Before concluding, related work is summarized in Section 5.

2 Background and Motivations

In this section, we briefly introduce modern memory architectures and how they affect application performance. Then we detail how existing software techniques try to overcome these issues and show the difficulty is to be as less intrusive as possible while trying to achieve high performance.

2.1 Modern Memory Architectures

The emergence of highly parallel architectures with many multicore processors raised the need to rethink the hardware memory subsystem. While the number of cores per machine quickly increases, memory performance remains several orders of magnitude slower. Concurrent accesses to central memory buses lead to dramatic contention, causing the overall performance to decrease. It led hardware designers to drop the centralized memory model in favor of distributed and hierarchical architectures, where memory nodes and caches are attached to some cores and far away from the others. This design has been widely used in high-end servers based on the ITANIUM processor. It now becomes mainstream since AMD HYPERTRANSPORT (see Figure 1) and the upcoming INTEL QUICKPATH memory interconnects dominate the server market. Indeed, these new memory architectures assemble multiple memory nodes into a single distributed cache-coherent system. It has the advantage of being as convenient to program as regular shared-memory SMP processors, while providing a much higher memory bandwidth and much less contention.

However, while being cache-coherent, these distributed architectures have non-constant physical distance between hardware components, causing their communication time to vary. Indeed, a core accesses local memory faster than other memory nodes. The ratio is often referred to as the *NUMA factor*. It generally varies from 1.2 up to 3 depending on the architecture and therefore has a strong impact on application performance. Not only does the application run faster if accessing local data, but also contention may appear on memory links if two processors access each others' memory nodes. Moreover, shared caches among cores increase the need to take data locality into account while scheduling tasks.

Table 1. Aggregated bandwidth on a quad-socket quad-core OPTERON machine depending on the machine load (4 or 16 threads) and the location of memory buffers

Data location	Local	Local + Neighbors
4 threads on node 0	5151 MB/s	5740 MB/s
4 threads per node (16 total)	4×3635 MB/s	4×2257 MB/s

To illustrate this problem, we ran some experiments on a quad-socket quad-core OPTERON machine. Second row of Table 1 shows that a custom application using few threads on a non-loaded machine will achieve best performance by distributing its pages across all memory nodes (to maximize the throughput) and keeping all threads together on a single processor (to benefit from a shared cache). However, on a loaded machine, having multiple threads access all memory nodes dramatically increases contention on memory links, thus achieving the best performance when each task only accesses local memory (third row of Table 1). This suggests that achieving high-performance on NUMA architecture takes more than just binding tasks and data based on their affinities. The host load and memory contention should also be taken into account.

2.2 Software Support for Memory Management

While the memory architecture complexity is increasing, the virtual memory model is slowly being extended to help applications achieving better performance.

Applications still manipulate virtual memory buffers that are mapped to physical pages that the system allocates anywhere on the machine. Most modern operating systems actually rely on a lazy allocation: when applications allocate virtual memory, the underlying physical pages are actually allocated upon the first access. While the primary advantage of this strategy is to decrease resource consumption, it brings an interesting feature usually referred to as *first-touch*: each page is allocated in the context of the thread that actually uses it first. The operating system is thus able to allocate physical pages on the memory node near the core that accesses it.

However, if the first thread touching a page is not the one that will access it intensively in the future, the page may not be allocated "in the right place". For this reason, some applications manually touch pages during the initialization phase to ensure that they are allocated close to the threads that will actually access them.

Dynamic applications such as adaptative meshes have their task/data affinities varying during the execution, causing the optimal distribution to evolve. One solution consists in migrating pages between memory nodes to move data near the tasks that access them at any time. However, it is expensive and requires actually to detect at runtime that a buffer is not located on the right node. Another solution called *next-touch* is the generalization of the *first-touch* approach: it allows applications to ask the system to allocate or migrate each page near the thread that will first touch it in the future [11,15,16]. It is unfortunately hard to implement efficiently and also does not work well in many cases, for instance if two threads are actually touching the same zone.

These features enable memory-aware task and data placement but they remain expensive. Moreover, as illustrated by the above experiment, predicting performance is difficult given that memory performance is also related to the machine load. Irregular applications will thus not only cause the thread scheduler to have to load-balance between idle and busy cores, but will also make the memory constraints vary dynamically, causing heuristics to become even harder to define.

3 Design of a Dynamic Approach to Place Threads and Memory

To tackle the problem of improving the overall application execution time over NUMA architectures, our approach is based on a flexible multi-level scheduling that continuously uses information about thread and data affinities.

3.1 Objectives

Our objective is to perform thread and memory placement dynamically according to some scheduling hints provided by the programmer, the compiler or even hardware counters. The idea is to map the parallel structure of the program onto the hardware architecture. This approach enables support for multiple strategies. At the machine level, the workload and memory load can be spread across

NUMA nodes and locality may be favored. All threads working on the same buffers may be kept together within the same NUMA node to reduce memory contention. At the processor level, threads that share data intensively may also be grouped to improve cache usage and synchronization [2]. Finally, inside multicore/multithreaded chips, access to independent resources such as computing units or caches may be taken into account. It offers the ability for a memory-intensive thread to run next to a CPU-intensive one without interference.

For irregular applications, all these decisions can only be taken at runtime. It requires an in-depth knowledge of the underlying architecture (memory nodes, shared caches, *etc.*). Our idea consists in using separate scheduling policies at various topology levels of the machine. For instance, low-level work stealing only applies to neighboring cores while the memory node level scheduler transfers larger entities (multiple threads with their data buffers) without modifying their internal scheduling. Such a transfer has to be decided at runtime after checking the hardware and application statuses. It requires that the runtime system remembers, during the whole execution, which threads are part of the same team and which memory buffers they often access. It should be possible to quantify these affinities as well as dynamically modify them if needed during the execution. To do so, affinity information may be attached at thread or buffer creation or later, either by the application programmer, by the compiler (through static analysis), or by the runtime system through instrumentation. In the end, the problem is to decide which actions have to be performed and when. We have identified the following events:

- When the application allocates or releases a resource (e.g., thread or buffer);
- When a processor becomes idle (blocking thread);
- When hardware counters reveal an issue (multiple accesses to remote nodes);
- When application programmers insert an explicit hint in their code.

To evaluate this model, we developed a proof-of-concept OpenMP extension based on instrumentation of the application. We now briefly present our implementation.

3.2 MaMI, a NUMA-Aware Memory Manager

MaMI is a memory interface implemented within our user-level thread library, Marcel [18]. It allows developers to manage memory with regard to NUMA nodes thanks to an automatically gathered knowledge of the underlying architecture. The initialization stage preallocates memory heaps on all the NUMA nodes of the target machine, and user-made allocations then pick up memory areas from the preallocated heaps.

MaMI implements two methods to migrate data. The first method is based on the *next-touch* policy, it is implemented as a user-level pager (`mprotect()` and signal handler for `SIGSEGV`). The second migration method is synchronous and allows to move data on a given node. Both move pages using the Linux system call `move_pages()`.

Migration cost is based on a linear function on the number of pages being migrated[1]. The cost in microseconds for our experimentation platform is $120+11\times nbpages$ (about 380 MB/s). It is also possible to evaluate reading and writing access costs to remote memory areas. Moreover, MaMI gathers statistics on how much memory is available and left on the different nodes. This information is potentially helpful when deciding whether or not to migrate a memory area. Table 2 shows the main functionalities provided by MaMI.

Table 2. Application programming interface of MaMI

- void ***mami_malloc**(memory_manager, size);
  Allocates memory with the default policy.
- int **mami_register**(memory_manager, buffer, size);
  Registers a memory area which has not been allocated by MaMI.
- int **mami_attach**(memory_manager, buffer, size, owner);
  Attaches the memory to the specified thread.
- int **mami_migrate_on_next_touch**(memory_manager, buffer);
  Marks the area to be migrated when next touched.
- int **mami_migrate_on_node**(memory_manager, buffer, node);
  Moves the area to the specified node.
- void **mami_cost_for_read_access**(memory_manager, source, dest, size, cost);
  Indicates the cost for read accessing SIZE bits from node SOURCE to node DEST.

3.3 ForestGOMP, a MaMI-Aware OpenMP Runtime

ForestGOMP is an extension to the GNU OpenMP runtime system relying on the Marcel/BubbleSched user-level thread library. It benefits from Marcel's efficient thread migration mechanism[2] thus offering control on the way OpenMP threads are scheduled. ForestGOMP also automatically generates groups of threads, called *bubbles*, out of OpenMP parallel regions to keep track of teammate threads relations *in a naturally continuous way.* Thanks to Marcel automatically gathering a deep knowledge of hardware characteristics such as cores, shared caches, processors and NUMA nodes, BubbleSched and MaMI are able to take interesting decisions when placing these bubbles and their associated data. The BubbleSched library provides specific *bubble schedulers* to distribute these groups of threads over the computer cores. The BubbleSched platform also maintains a programming interface for developing new bubble schedulers. For example, the *Cache* bubble scheduler [2] has been developed using this interface. Its main goal is to benefit from a good cache memory usage by scheduling teammate threads as close as possible on the computer and

[1] Our experiments first showed a non-linear behavior for the migration cost. It led us to improve the `move_pages()` system call implementation in Linux kernel 2.6.29 to reduce the overhead when migrating many pages at once [8].

[2] The thread migration cost is about 0.06μs per thread and the latency is about 2.5μs.

stealing threads from the most local cores when a processor becomes idle. It also keeps track of where the threads were being executed when it comes to perform a new thread and bubble distribution.

The ForestGOMP platform has been enhanced to deal with memory affinities on NUMA architectures.

A Scheduling Policy Guided by Memory Hints. Even if the *Cache* bubble scheduler offers good results on dynamic cache-oblivious applications, it does not take into account memory affinities, suffering from the lack of information about the data the threads access. Indeed, whereas keeping track of the bubble scheduler last distribution to move threads on the same core is not an issue, the BubbleSched library needs feedback from the memory allocation library to be able to draw threads and bubbles to their *"preferred"* NUMA node. That is why we designed the *Memory* bubble scheduler that relies on the MaMI memory library to distribute threads and bubbles over the NUMA nodes regarding their memory affinities. The idea here is to have MaMI attaching *"memory hints"* to the threads by calling the BubbleSched programming interface. These hints describe how much data a thread will use, and where the data is located. This way, the bubble scheduler is able to guide the thread distribution onto the correct NUMA nodes. Then, the *Cache* bubble scheduler is called inside each node to perform a cache-aware distribution over the cores.

Extending ForestGOMP to Manage Memory. The ForestGOMP platform has also been extended to offer the application programmer a new set of functions to help convey memory-related information to the underlying OpenMP runtime. There are two main ways to update this information. Application programmers can express memory affinities by the time a new parallel region is encountered. This allows the ForestGOMP runtime to perform early optimizations, like creating the corresponding threads at the right location. Updating memory hints inside a parallel region is also possible. Based on these new hints, the bubble scheduler may decide to redistribute threads. Applications can specify if this has to be done each time the updating function is called, or if the runtime has to wait until all the threads of the current team have reached the updating call. The ForestGOMP runtime only moves threads if the new per-thread memory information negates the current distribution.

4 Performance Evaluation

We first describe in this section our experimentation platform and we detail the performance improvements brought by ForestGOMP on increasingly complex applications.

4.1 Experimentation Platform

The experimention platform is a quad-socket quad-core 1.9 GHz Opteron 8347HE processor host depicted on Figure 1. Each processor contains a 2MB shared L3 cache and has 8 GB memory attached.

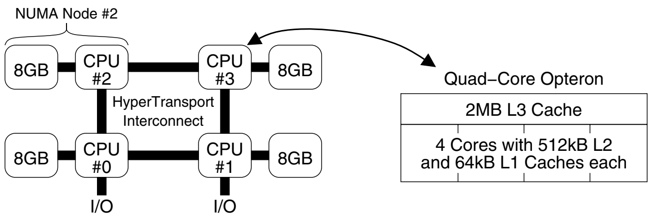


Fig. 1. The experimentation host is composed of 4 quad-core OPTERON (4 NUMA nodes)

Table 3. Memory access latency (uncached) depending on the data being local or remote

Access type	Local access	Neighbor-node access	Opposite-node access
Read	83 ns	98 ns (× 1.18)	117 ns (× 1.41)
Write	142 ns	177 ns (× 1.25)	208 ns (× 1.46)

Table 3 presents the NUMA latencies on this host. Low-level remote memory accesses are indeed much slower when the distance increases. The base latency and the NUMA factor are higher for write accesses due to more hardware traffic being involved. The observed NUMA factor may then decrease if the application accesses the same cache line again as the remote memory node is not involved synchronously anymore. For a write access, the hardware may update the remote memory bank in the background (*Write-Back Caching*). Therefore, the NUMA factor depends on the application access patterns (for instance their spatial and temporal locality), and the way it lets the cache perform background updates.

4.2 STREAM

STREAM [12] is a synthetic benchmark developed in C, parallelized using OpenMP, that measures sustainable memory bandwidth and the corresponding computation rate for simple vectors. The input vectors are wide enough to limit the cache memory benefits (20 millions double precision floats), and are initialized in parallel using a *first-touch* allocation policy to get the corresponding memory pages close to the thread that will access them.

Table 4 shows the results obtained by both GCC 4.2 LIBGOMP and FORESTGOMP runtimes running the STREAM benchmark. The LIBGOMP library exhibits varying performance (up to 20%), which can be explained by the fact the underlying kernel thread library does not bind the working threads on the computer cores. Two threads can be preempted at the same time, and switch their locations, inverting the original memory distribution. The FORESTGOMP runtime achieves a very stable rate. Indeed, without any memory information, the *Cache* bubble scheduler deals with the thread distribution, binding them to the cores. This way, the *first-touch* allocation policy is valid during the whole application run.

Table 4. STREAM benchmark bandwidth results on a 16-core machine (1 thread per core), in MB/s

	LIBGOMP	FORESTGOMP
Operation	Worst-Best	Worst-Best
Copy	6 747-8 577	7 851-7 859
Scale	6 662-8 566	7 821-7 828
Add	7 132-8 821	8 335-8 340
Triad	7 183-8 832	8 357-8 361

4.3 Nested-STREAM

To study further the impact of thread and data placement on the overall application performance, we modified the STREAM benchmark program to use nested OpenMP parallel regions. The application now creates one team per NUMA node of the computer. Each team works on its own set of STREAM vectors, that are initialized in parallel, as in the original version of STREAM. To fit our target computer architecture, the application creates four teams of four threads. Table 5 shows the results obtained by both the LIBGOMP and the FORESTGOMP library.

The LIBGOMP runtime system maintains a pool of threads for non-nested parallel regions. New threads are created each time the application reaches a nested parallel region, and destroyed upon work completion. These threads can be executed by any core of the computer, and not necessarily where the master thread of the team is located. This explains why the results show a large deviation.

The FORESTGOMP runtime behaves better on this kind of application. The underlying bubble scheduler distributes the threads by the time the outer parallel region is reached. Each thread is permanently placed on one NUMA node of the computer. Furthermore, the FORESTGOMP library creates the teammates threads where the master thread of the team is currently located. As the vectors accessed by the teammates have been touched by the master thread, this guarantees the threads and the memory are located on the same NUMA node, and thus explains the good performance we obtain.

Table 5. Nested-STREAM benchmark bandwidth results in MB/s

	LIBGOMP	FORESTGOMP
Operation	Worst-Best	Worst-Best
Copy	6 900-8 032	8 302-8 631
Scale	6 961-7 930	8 201-8 585
Add	7 231-8 181	8 344-8 881
Triad	7 275-8 123	8 504-9 217

4.4 Twisted-STREAM

To complicate the STREAM memory access pattern, we designed the Twisted-STREAM benchmark application, which contains two distinct phases. The first one behaves exactly as Nested-Stream, except we only run the Triad kernel here, because it is the only one to involve the three vectors. During the second phase, each team works on a different data set than the one it was given in the first phase. The *first-touch* allocation policy only gives good results for the first phase as shown in Table 6.

Table 6. Average rates observed with the Triad kernel of the Twisted-Stream benchmark using a *first-touch* allocation policy. During phase 2, threads access data on a different NUMA node.

	libgomp	ForestGOMP
Triad Phase 1	8 144 MB/s	9 108 MB/s
Triad Phase 2	3 560 MB/s	6 008 MB/s

A typical solution to this lack of performance seems to rely on a *next-touch* page migration between the two phases of the application. However this functionality is not always available. And we show in the remaining of this section that the *next-touch* policy is not always the best answer to the memory locality problem.

The Stream benchmark program works on three 160MB-vectors. We experimented with two different data bindings for the second phase of Twisted-Stream. In the first one, all three vectors are accessed remotely, while in the second one, only two of them are located on a remote node. We instrumented both versions with calls to the ForestGOMP API to express which data are used in the second phase of the computation.

Remote Data. The underlying runtime system has two main options to deal with remote accesses. It can first decide to migrate the three vectors to the NUMA node hosting the accessing threads. It can also decide to move the threads

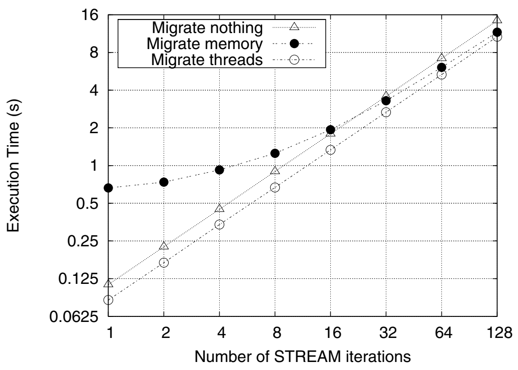


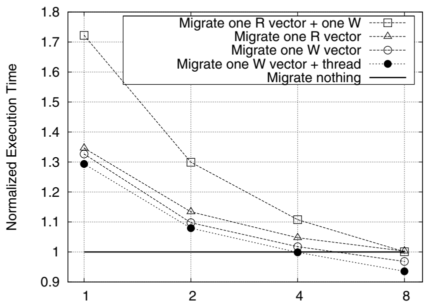
Fig. 2. Execution times of different thread and memory policies on the Twisted-Stream benchmark, where the whole set of vectors is remotely located

to the location of the remote vectors. Figure 2 shows the results obtained for both cases.

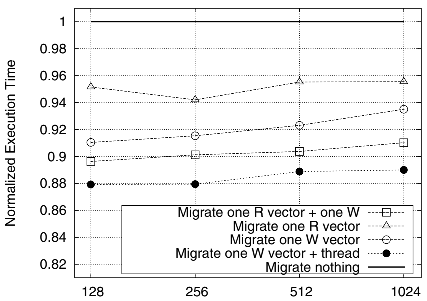
Moving the threads is definitely the best solution here. Migrating 16 threads is faster than migrating the corresponding vectors, and guarantees that every team only accesses local memory. On the other hand, if the thread workload becomes big enough, the cost for migrating memory may be become lower than the cost for accessing remote data.

Mixed Local and Remote Data. For this case, only two of the three STREAM vectors are located on a remote NUMA node. One of them is read, while the other one is written. We first study the impact of the NUMA factor by only migrating one of the two remote vectors. Figure 3(a) shows the obtained performance. As mentioned in Table 3, remote read accesses are cheaper than remote write accesses on the target computer. Thus, migrating the read vector is less critical, which explains our better results when migrating the written vector. The actual performance difference between migrating read and written vectors is due to twice as many low-level memory accesses being required in the latter case.

To obtain a better thread and memory distribution, the underlying runtime can still migrate both remote vectors. Moving only the threads would not discard the remote accesses as all three vectors are not on the same node. That is why we propose a mixed approach in which the FORESTGOMP runtime system migrates both thread and local vector near to the other vector. This way, since migrating threads is cheap, we achieve a distribution where all the teams access their data locally while migrating as few data as possible. Figure 3(a) shows the overhead of this approach is smaller than the *next-touch* policy, for which twice as much data is migrated, while behaving the best when the thread workloads increase, as we can see on Figure 3(b).



(a) First iterations



(b) Further iterations

Fig. 3. Execution times of different thread and memory policies on the Twisted-STREAM benchmark, where only two of the three vectors are remotely located

We also tested all our three STREAM benchmark versions on the Intel compiler 11.0, which behaves better than FORESTGOMP on the original STREAM application (10 500 MB/s) due to compiler optimizations. Nevertheless, performance drops significantly on both Nested-Stream, with an average rate of 7 764 MB/s, and Twisted-Stream with a second step average rate of 5 488 MB/s, while the ForestGOMP runtime obtains the best performance.

5 Related Work

Many research projects have been carried out to improve data distribution and execution of OpenMP programs on NUMA architectures. This has been done either through HPF directives [1] or by enriching OpenMP with data distribution directives [4] directly inspired by HPF and the SGI Fortran compiler. Such directives are useful to organize data the right way to maximize page locality, and, in our research context, a way to transmit affinity information to our runtime system without heavy modifications of the user application.

Nikolopoulos *et al.* [13] designed a mechanism to migrate memory pages automatically that relies on user-level code instrumentation performing a sampling analysis of the first loop iterations of OpenMP applications to determine thread and memory affinity relations. They have shown their approach can be even more efficient when the page migration engine and the operating system scheduler [14] are able to communicate. This pioneering research only suits OpenMP applications that have a regular memory access pattern while our approach favors many more applications.

To tackle irregular algorithms, [11,15,16] have studied the promising *next-touch* policy. It allows the experienced programmer to ask for a new data distribution explicitly the next time data is touched. While being mostly similar to the easy-to-use *first-touch* memory placement policy in terms of programming effort, the *next-touch* policy suffers from the lack of cooperation between the allocation library and the thread scheduler. Moreover, this approach does not take the underlying architecture into account and so can hardly achieve most of its performance. ForestGOMP works around this issues thanks to BubbleSched and MaMI knowledge of the underlying processor and memory architecture and load.

6 Conclusion and Future Work

Exploiting the full computational power of current more and more hierarchical multiprocessor machines requires a very careful distribution of threads and data among the underlying non-uniform architecture. Directive-based programming languages provide programmers with a portable way to specify the parallel structure of their application. Through this information, the scheduler can take appropriate load balancing decisions and either choose to migrate memory, or decide to move threads across the architecture. Indeed, thread/memory affinity does matter mainly because of congestion issues in modern NUMA architectures.

Therefore, we introduce a multi-level thread scheduler combined with a NUMA-aware memory manager. It enables dynamic load distribution in a coherent way based on application requirements and hardware constraints, thus helping to reach performance portability. Our early experiments show that mixed solutions (migrating threads and data) improve overall performance. Moreover, traditional *next-touch*-based data distribution approaches are not always optimal since they are not aware of the memory load of the target node. Migrating threads may be more efficient in such situations.

We plan first to enhance the current bubble framework so as to improve our scheduling decision criteria by introducing global redistribution phases at some times. Obviously, hardware counter feedback should also be involved in this process. We therefore need to experiment further with both synthetic and real-life applications.

These results also suggest there is a need to extend OpenMP so as to transmit task/memory affinity relations to the underlying runtime. This evolution could also widen the OpenMP spectrum to hybrid programming [5,6].

References

1. Benkner, S., Brandes, T.: Efficient parallel programming on scalable shared memory systems with High Performance Fortran. In: Concurrency: Practice and Experience, vol. 14, pp. 789–803. John Wiley & Sons, Chichester (2002)
2. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.-A.: Scheduling Dynamic OpenMP Applications over Multicore Architectures. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 170–180. Springer, Heidelberg (2008)
3. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, George Mason University (May 1999)
4. Chapman, B.M., Bregier, F., Patil, A., Prabhakar, A.: Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems. In: Concurrency: Practice and Experience, vol. 14, pp. 713–739. John Wiley & Sons, Chichester (2002)
5. Dolbeau, R., Bihan, S., Bodin, F.: HMPP$^{\text{TM}}$: A Hybrid Multi-core Parallel Programming Environment. Technical report, CAPS entreprise (2007)
6. Duran, A., Perez, J.M., Ayguade, E., Badia, R., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependant Tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada (June 1998)
8. Goglin, B., Furmento, N.: Enabling High-Performance Memory-Migration in Linux for Multithreaded Applications. In: MTAAP 2009: Workshop on Multithreaded Architectures and Applications, held in conjunction with IPDPS 2009, Rome, Italy, May 2009. IEEE Computer Society Press, Los Alamitos (2009)
9. Intel. Thread Building Blocks, `http://www.intel.com/software/products/tbb/`
10. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook (1994)
11. Löf, H., Holmgren, S.: Affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system. In: 19th ACM International Conference on Supercomputing, Cambridge, MA, USA, June 2005, pp. 387–392 (2005)

12. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995, pp. 19–25 (1995)
13. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E.: User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In: International Conference on Parallel Processing, September 2000, pp. 95–103. IEEE Computer Society Press, Los Alamitos (2000)
14. Nikolopoulos, D.S., Polychronopoulos, C.D., Papatheodorou, T.S., Labarta, J., Ayguadé, E.: Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors. Parallel and Distributed Computing 62, 1069–1103 (2002)
15. Nordén, M., Löf, H., Rantakokko, J., Holmgren, S.: Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 382–393. Springer, Heidelberg (2008)
16. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: MAW 2008: Proceedings of the 2008 workshop on Memory access on future processors, pp. 377–384. ACM, New York (2008)
17. Thibault, S., Broquedis, F., Goglin, B., Namyst, R., Wacrenier, P.-A.: An efficient openMP runtime system for hierarchical architectures. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 161–172. Springer, Heidelberg (2008)
18. Thibault, S., Namyst, R., Wacrenier, P.-A.: Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 42–51. Springer, Heidelberg (2007)

Scalability of Gaussian 03 on SGI Altix: The Importance of Data Locality on CC-NUMA Architecture

Roberto Gomperts[1], Michael Frisch[2], and Jean-Pierre Panziera[1]

[1] Silicon Graphics, Inc., 1140 E. Arques Ave.
Sunnyvale, CA 94085, USA
[2] Gaussian, Inc., 340 Quinnipiac St Bldg 40
Wallingford, CT 06492, USA
`roberto@sgi.com`,
`frisch@gaussian.com`,
`jpp@sgi.com`

Abstract. Performance anomalies when running Gaussian frequency calculations in parallel on SGI Altix computers with CC-NUMA memory architecture are analyzed using performance tools that access hardware counters. The bottleneck is the frequent and nearly simultaneous data-loads of all threads involved in the calculation of data allocated in the node where the master thread runs. Code changes that ensure these data-loads are localized improve performance by a factor close to two. The improvements carry over to other molecular models and other types of calculations. An expansion or an alternative of `FirstPrivate` OpenMP's clause can facilitate the code transformations.

Keywords: OpenMP, Gaussian 03, SGI Altix, CC-NUMA, memory latency.

1 Introduction

The Computational Chemistry program Gaussian [1] is one of the few commercially available programs in the discipline that has been parallelized using OpenMP [2].

When monitoring the progress of a 32-way parallel Gaussian 03 frequency calculation on an SGI Altix 450 with the system command "top," we noticed an anomalous performance behavior: The first four threads clearly lagged behind the rest. This effect is most visible in the section of Gaussian that solves the Couple-Perturbed Hartree-Fock (CPHF) equations [3], [4], [5], [6] (link 11002). This lack of load balance cannot be attributed to either the parallel algorithm in Gaussian or to possible peculiarities of the input data set.

In this paper we describe first the Gaussian software, followed by a brief explanation of the SGI Altix CC-NUMA (cache coherent Non-Uniform Memory Access) architecture, and of the profiling tools that enable the access to hardware counters. In the next sections we discuss the possible origin of the interactions with the CC-NUMA memory architecture of the SGI Altix 450 and a way to overcome them. In closing we offer a suggestion of an OpenMP clause that would make implementing the solution straightforward.

2 Gaussian

2.1 The Gaussian 03 Program

Gaussian 03 computes approximate solutions to the Schrödinger Equation for molecular systems. The Integro-differential equations for the electronic wavefunction for a given arrangement of the nuclei is approximated by expansion in a finite basis of Gaussian functions [7] which reduces the problem to linear algebra plus the evaluation of matrix elements involving these basis functions. The calculations performed in this work involve a mean-field approximation to the general Schrödinger equation [8], [9] using a density-functional expression for the mean-field potential [10].

The number of basis functions, N, is proportional to the size of the molecule (number of atoms, N_A). The number of matrix elements is formally $O(N^4)$, but because the basis functions are localized in space, after applying numerical thresholds this number approaches $O(N^2)$ for large molecules. The cost of using the matrix elements can be reduced to O(N) using tree-based methods such as the Fast Multipole Method [11], [12] for the Coulomb part of the problem and appropriate use of the locality of interactions for other terms in the potential [13], [14], [15].

Stationary points on the potential energy surface for nuclear motion are located using standard optimization techniques along with analytic first derivatives of the electronic energy with respect to the nuclear coordinates [16]. Minima on a surface correspond to stable structures of the molecules. Second derivatives of the energy with respect to the nuclear coordinates at minima can be used to predict harmonic vibrational frequencies and vibrational spectra, such as Infra-Red (IR) and Vibrational Circular Dichroism (VCD) [17]. Since the analytic second derivative calculation can be very time-consuming, one of these calculations is used here as an example for testing the performance of various algorithms with respect to memory access in a CC-NUMA environment.

Time spent in the second derivative calculation is dominated by solving the derivative self-consistent field (CPHF) equations. There is one such equation for each nuclear displacement, for a total of $3N_A$. The dimensionality of these equations is large, so they are solved iteratively, with each iteration involving the contraction of matrix elements with contributions to each derivative density. Thus the computational cost is formally $O(N_A N^4)$. The scaling can be reduced for large systems as mentioned above, but for the modest size system used as an example here, the resulting speedups are modest.

2.2 Memory Allocation and Parallelism in Gaussian

Many of the terms in the calculations done in Gaussian involve very large amounts of data. Most of these are recomputed as needed. But since it is not possible to store all the data (e.g., all the N^4 matrix elements) in memory, the management of memory and the selection of algorithms based on the amount of available memory is critical to good performance. Consequently, a stack-based allocation approach is used. The amount of memory to be used by the calculation is set by the user and the memory allocated once. Then routines which compute individual terms in the calculation choose their algorithms and allocate memory from this pool, passing on unused parts for use by the subordinate routines for the chosen algorithm. This facilitates a strategic approach in which the optimal method of calculation given the amount of memory available can be selected, but it also means that on a CC-NUMA system with a "first-touch" memory allocation policy, the physical location of pages is determined by whatever term was calculated first, and this need not be optimal for other phases of the overall calculation.

The underlying algorithms for basic tasks, such as the computation of matrix elements, are frequently updated and improved. So the primary goal in the implementation of parallelism in Gaussian is to have a single set of routines which implement an algorithm and which can be used for serial, SMP parallel, and cluster parallel computations. SMP parallelism is implemented using OpenMP and cluster parallelism, which is not considered further in this paper, is done using Linda. In the example computation considered here, the dominant step is computation of the N^4 two-electron integrals and their contraction with density matrix derivative contributions during solution of the CPHF equations. This is parallelized by dividing the integrals to be computed among threads, with each thread computing the contribution of some integrals to all the products. Since the OpenMP clause `Default(Shared)` is used in every `Parallel Region` and `Parallel Do`, a copy of the output products is allocated to each thread, and the results are added together after all threads have completed their tasks. Each thread also has space allocated for the intermediate data generated and used during the calculation, but all threads share the input data (density matrices and various quantities derived from the basis functions which are used in the computation of the integrals). The allocation of work is done statically so that no communication between tasks is required. For the example considered here, static allocation gives excellent load balance, but it is also trivial to distribute work dynamically via a shared counter if this is preferable for other cases.

2.3 Example Calculation

Our example is a calculation of the IR and VCD spectra of alpha-pinene, first published in ref [18]. The calculation done here uses the same B3LYP model as the original work, but the 6-311G(df,p) basis set. This is a modest size of calculation by modern standards, but this facilitates testing a variety of modifications to the algorithms and memory layout. The performance issues involved in larger calculations are unchanged from this example.

3 SGI Altix

SGI Altix systems are based on a cache-coherent Non Uniform Memory Access (CC-NUMA) architecture. The system components (processors, memory, IO, etc.) are distributed across nodes. These nodes are interconnected through an internal network. The cache coherency is maintained through a directory located in each memory node. Reference [19] describes the principle of this type of Distributed Shared Memory (DSM) systems.

On SGI Altix systems, the "compute" nodes include two processor sockets, memory DIMMs (dual in-line memory modules) and a "Hub" chip. The "Hub" is the memory controller for the local DIMMs. It also interfaces with the processors through the Front Side Bus (FSB) and the rest of the system with two NUMAlink ports. The NUMAlink fabric interconnects the different nodes in the system through eight port routers. Each link sustains a bandwidth of 3.2 GB/s in each direction. It is also possible to configure a system with memory-only nodes.

The experiments described in this paper were carried out on an SGI Altix 450 system [20] equipped with 9130M Dual-Core Intel Itanium Processors. These processors run at 1.66GHz and have a last level cache size of 8MB (4 MB per core). The FSB has a frequency of 667 MHz. The computer system has 72 cores and 2GB of memory per core. The architecture of this system is sketched on Fig. 1; each of the two symmetric NUMAlink fabrics is formed by four routers in a square attached to up to 5 nodes (Compute or IO). The Non-Uniform aspect of the CC-NUMA architecture can be easily understood from this picture. Access to the local memory simply goes through the local “hub,” while a remote access will require another “hub” and one, two or three router traversals. As a result, the remote memory-load latency is between 2.1 and 2.9 times larger than the local direct access.

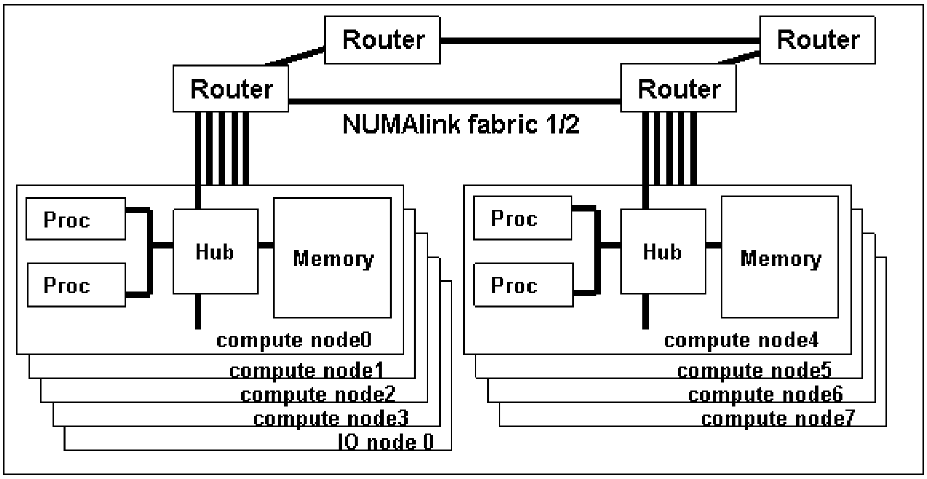


Fig. 1. SGI Altix 450 architecture. Only one NUMAlink interconnection represented. The system may include up to 80 nodes, with at least one IO and one Compute node.

4 Performance Tools

For this study, we used the SGI Histx [20] performance analysis tools, which SGI developed for the IA64 architecture. The SGI Histx performance tool suite includes two tools used in this project: a profiling tool *histx,* and a performance monitoring tool *lipfpm*. *lipfpm* simply counts the occurrence of given events during a run; it relies on the hardware performance counters [22] available on the Intel Itanium processors. The profiling tool *histx* can also use these performance counters to profile an application and monitor the frequency of any event for each section of the code.

In particular for this project we used the following experiments:

- *lipfpm "mlat"* computes the average memory access latency seen at the processor interface (FSB) dividing the cumulative number of memory read transactions outstanding per cycle by the total number of cycles.
- *lipfpm "dlatNN"* simply counts the number of memory data-loads which took more than NN cycles to complete; possible values for NN are any power of two between 4 and 4096. E.g. *lipfpm "dlat1024"* measures the number of data-loads which took more than 1024 cycles.
- *histx "dlatNN"* profiles the application based on the occurrence of data-loads with latency longer than NN cycles. E.g. with *histx "dlat1024@2000"* a sample is taking after 2000 data-loads longer than 1024 cycles. So if an application generates 200 million data-loads longer than 1024 cycles, 100 thousand samples will be taken allowing for a precise profile.
- *histx* "*numaNN*" also samples data-loads longer than NN cycles (like histx "*dlatNN*" does). However it further identifies which node of the CC-NUMA system the data accessed was located on.

Note that the *histx* profiling experiments provide information at the routine or the source line level.

5 Discussion

The model job used in this paper is the calculation of vibrational frequencies of α-pinene. This molecule consists of 26 atoms. The basis set, a measure of the accuracy of the calculation, is 6-311G(df,p); this results in a total of 346 basis functions which is an indication of the size of the problem. The full calculation executed on 32 cores of the previously described SGI Altix 450 takes around 1450 seconds wall clock time. The section of the program where the anomalous behavior was noticed, link l1002, takes 1144 seconds; this is 79% of the time for the full calculation.

The experiments to measure latencies using the original code confirm that the first 4 threads that are located on the first blade (node 0) have much lower average latency than those threads on other nodes. We will call them remote. These effects are shown in Fig. 2 and Fig. 3.

Fig. 2 depicts the results of the *"mlat" lipfpm* experiment. Thread 0, the master thread, has an Average Memory Latency of around 480 ns as seen by the FSB. The average of the Average Memory Latency of the next threads, located on the same node, is slightly lower, around 470 ns. Unlike thread 0 these threads do not have to deal with gather operations. The next 28 threads exhibit on average a much higher Average Memory Latency, 955 ns, almost a factor of two compared to thread 0.

Similar effects can be seen in Fig. 3. For thread 0 there are around 20 million retired data loads with latencies that exceed 1024 cycles (613 ns). The threads on the same node count less than 5% of such data loads, whereas the average of the counts for the remaining threads is close to 10 times higher.

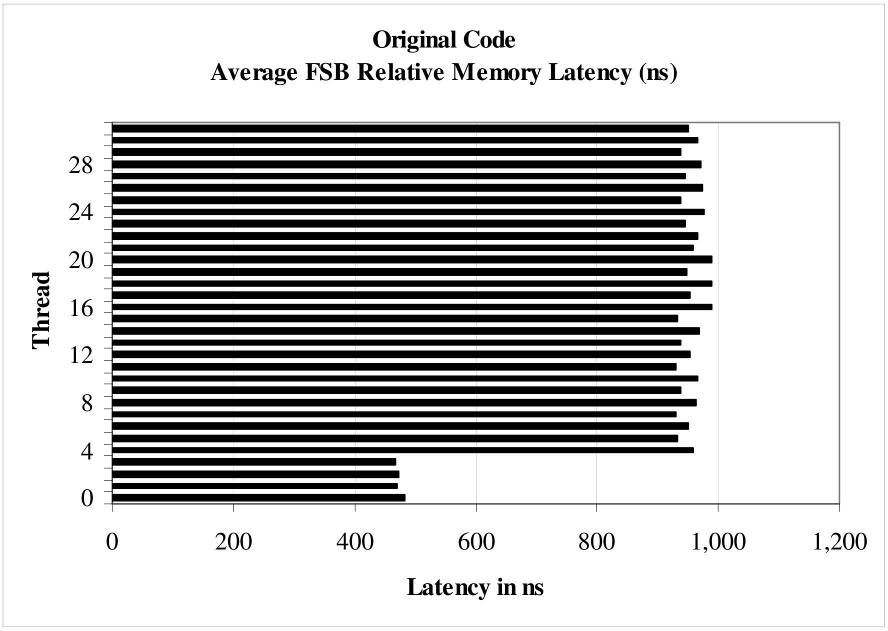


Fig. 2. Result of the "*mlat*" *lipfpm* experiment using the original code. This figure shows the Average FSB Memory Latency in nanoseconds for every thread in the calculation: 0 to 31.

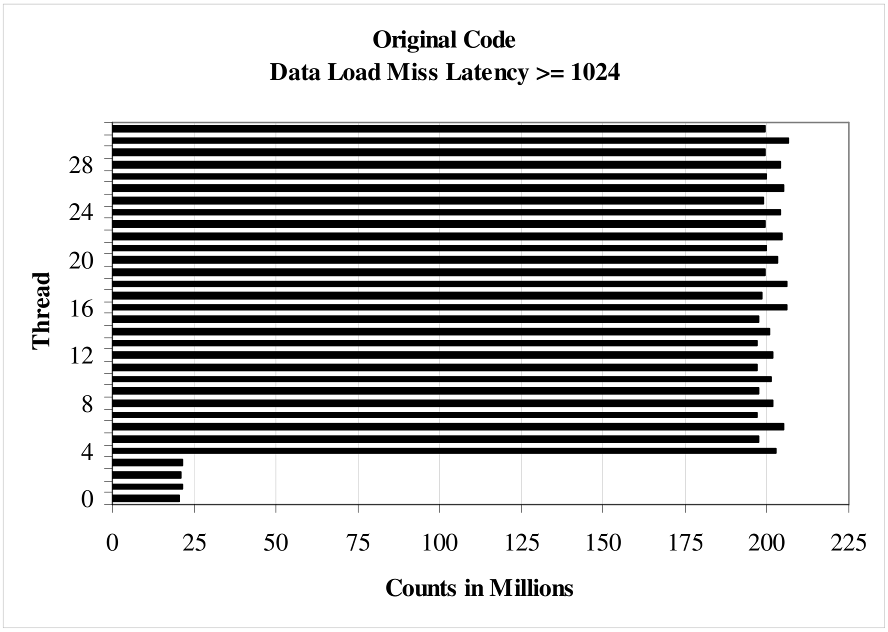


Fig. 3. Result of "*dlat1024*" *lipfpm* experiment using the original code. This experiment counts the number of data-load misses that take more than 1024 cycles for each thread 0 to 31.

Experiments with the *histx* tool ("*numa*") point directly to the function in the Gaussian program that is responsible for the vast majority of the long-latency data-loads, `dgst01`. In particular we can see that these data-loads are coming from accesses to the (remote) node where thread 0 is running.

The "hot-spot" of `dgst01` has constructions like the one shown in Table 1.

Table 1. Program snippet from dgst01

```
Do 500 IShC = 1, NShCom
   IJ = LookLT(C4IndR(IShC,1)+IJOff)
   KL = LookLT(C4IndR(IShC,6)+KLOff)
   R1IJKL = C4RS(IShC,IRS1,1)*( C4ERI(IShC,IJKL)
$          - FactX*(C4ERI(IShC,IKJL)+C4ERI(IShC,ILJK)) )
C
   If(Abs(R1IJKL).ge.CutOff) then
      Do 10 IMat = 1, NMatS
10       FA(IMat,IJ) = FA(IMat,IJ) + DA(IMat,KL)*R1IJKL
      Do 20 IMat = 1, NMatS
20       FA(IMat,KL) = FA(IMat,KL) + DA(IMat,IJ)*R1IJKL
   endIf

  [Similar If/endIf constructs as above]

500   Continue
```

The arrays of interest are `FA`, the Fock matrix, and `DA`, the Density matrix. These are used in the inner loops 10 and 20. These loops have an overall nesting of 5.

Initially we focused our attention on `FA` thinking that the load/store (read/write) activity would be the main source of the long-latency data load misses. Even though this array should have been mostly localized on the node where the thread is running, we allowed for the possibility that this localization may not be "perfect" due to possible accesses to the memory locations in the serial part or a previous parallel region.

We ensured localization by creating ALLOCATABLE temporary arrays that where allocated and first used in the parallel region, thus avoiding the potential problems of previously allocated and used memory from the stack managed by Gaussian. This code modification did not have any effect on the Average Memory Latency of the remote threads nor on the counts of long-latency data-loads.

We therefore turned our attention to DA. After implementing the same modifications as for `FA`, we saw a very substantial improvement in the latency profile of the memory accesses. Figures 4 and 5 summarize the results.

The first thing to notice in Fig. 4 is that all average latencies are lower. The latencies in the first node have come down by approximately 100 ns to 360 (thread 0) and 320 ns (average of threads 1 to 3). But more importantly, the average latency of the remote nodes has been more than halved -- now 370 ns on average. This means that it is of the same order of magnitude as the latency of thread 0. Consequently the observed load balance is improved significantly.

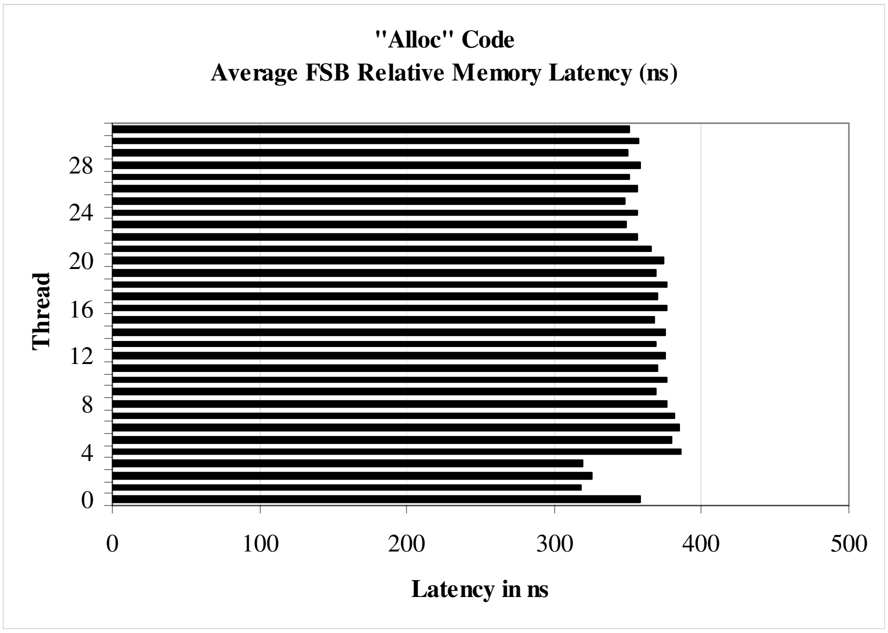


Fig. 4. Result of the "*mlat*" *lipfpm* experiment using the modified code. This figure shows the Average FSB Memory Latency in nanoseconds for every thread in the calculation: 0 to 31.

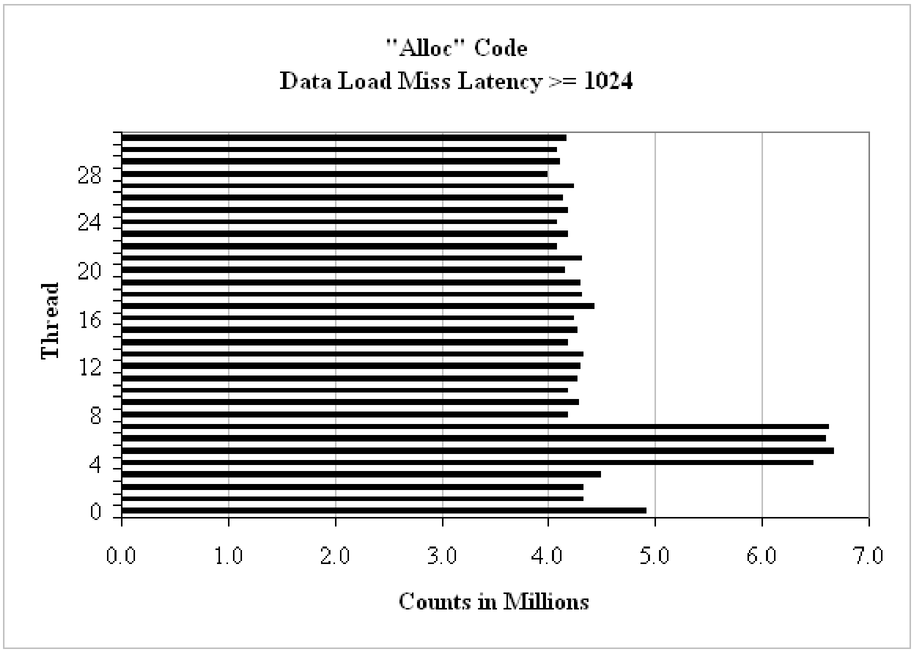


Fig. 5. Result of "*dlat1024*" *lipfpm* experiment using the modified code. This experiment counts the number of data-load misses that take more than 1024 cycles for each thread 0 to 31.

This is also reflected in Fig. 5 to be compared with Fig. 3. The remote long-latency data-load counts have been cut by a factor of around 5 on the first node and by a factor of over 40 on the other nodes. We don't have a good explanation for the higher latencies in the second node, but it should be noted that these differences are negligible in comparison with the gains obtained from the data in Fig. 3. In addition, subsequent experiments did show a uniform distribution of the latencies of all the remote nodes. Higher in the profile are now routines that take care of data movement (copies).

The improved locality and derived decrease in latency has an important impact on the overall performance of this section of the calculation. The modified code runs this portion in 668 seconds elapsed time, down from 1144 seconds needed by the original program.

The original code represents the worst-case scenario for contention since all the "remote" nodes are trying to access elements of `DA` that have been allocated in the first node. In this sense the modified code exemplifies the best case scenario since all the references to `DA` are local. An example of an intermediate case is when the array `DA` is distributed uniformly over the nodes. In this case, while there is no strict locality for `DA`, the data-load contention is spread over the network fabric. As a consequence no load unbalance is seen. The Average FSB Memory Latency of thread 0 remains unchanged compared to the original code. The average for the threads in the first node is just a bit lower than in the original code. The biggest improvement is, as to be expected, in the average for the "remote" nodes. It improves by close to a factor of 2 to 490 ns. The run time for this version is around 800 sec., a significant improvement compared to the original code but not nearly as good as for the modified code. These results are insensitive to how `DA` is distributed over the nodes: equal, contiguous chunks over the threads or round-robin with pagesize segments.

We have verified that similar improvements can be measured when running the same type of calculation on different molecules. The benefits extend to other types of Gaussian calculations where the same core algorithm is used.

6 Suggestions

The effects of the code modifications could have largely been accomplished using the existing OpenMP clause `FirstPrivate`. This clause, however, creates `n` copies (where `n` is the number of threads) of the array whereas only `n-1` (or less, see below) are needed in the case where the private arrays are read-only. It also lacks the flexibility to address specific situations.

A new clause that has at least the following properties would be desirable:

- The original array in the master thread should not be replicated. As an advanced feature the option can be given to not replicate the arrays for the threads running in the first node.
- There is no need for a `Barrier` in the allocation of these read-only private arrays
- A conditional clause would be desirable. A decision about replicating individual arrays should be made at run time.
- Run time determination of size and shape of arrays is also desirable. This would help to work around arrays with assumed dimensions and or shapes. Cases where only sections of the array are needed in the parallel region would also be addressed.

7 Conclusions

In CC-NUMA systems with default “first-touch” memory allocation policies, frequent access to memory locations in remote nodes can have a profound impact on performance. In the case presented in this paper, the culprit is the significant difference in latency between local and remote data-loads, especially when all remote data-loads are from one node. It is fairly simple to recode the program to attenuate these effects, but this goes at the expense of a larger memory footprint. The programming task can be made “OpenMP-friendly” with a special clause to the parallel region directives.

The need to localize by replication read-only memory locations is not intuitive in OpenMP programming. In general, with a default setting of sharing variables in a parallel region, the focus is directed towards privatizing read-write data to minimize or avoid altogether the use of critical regions.

As more and larger multi-core microprocessors are being developed the considerations presented in this paper become increasingly relevant. An efficient shared memory application must take into account the high cost of remote access. The programmer should not hesitate and replicate data structures frequently accessed.

References

1. Frisch, M.J., Trucks, G.W., Schlegel, H.B., Scuseria, G.E., Robb, M.A., Cheeseman, J.R., Montgomery Jr., J.A., Vreven, T., Scalmani, G., Mennucci, B., Barone, V., Petersson, G.A., Caricato, M., Nakatsuji, H., Hada, M., Ehara, M., Toyota, K., Fukuda, R., Hasegawa, J., Ishida, M., Nakajima, T., Honda, Y., Kitao, O., Nakai, H., Li, X., Hratchian, H.P., Peralta, J.E., Izmaylov, A.F., Kudin, K.N., Heyd, J.J., Brothers, E., Staroverov, V.N., Zheng, G., Kobayashi, R., Normand, J., Sonnenberg, J.L., Ogliaro, F., Bearpark, M., Parandekar, P.V., Ferguson, G.A., Mayhall, N.J., Iyengar, S.S., Tomasi, J., Cossi, M., Rega, N., Burant, J.C., Millam, J.M., Klene, M., Knox, J.E., Cross, J.B., Bakken, V., Adamo, C., Jaramillo, J., Gomperts, R., Stratmann, R.E., Yazyev, O., Austin, A.J., Cammi, R., Pomelli, C., Ochterski, J.W., Ayala, P.Y., Morokuma, K., Voth, G.A., Salvador, P., Dannenberg, J.J., Zakrzewski, V.G., Dapprich, S., Daniels, A.D., Strain, M.C., Farkas, O., Malick, D.K., Rabuck, A.D., Raghavachari, K., Foresman, J.B., Ortiz, J.V., Cui, Q., Baboul, A.G., Clifford, S., Cioslowski, J., Stefanov, B.B., Liu, G., Liashenko, A., Piskorz, P., Komaromi, I., Martin, R.L., Fox, D.J., Keith, T., Al-Laham, M.A., Peng, C.Y., Nanayakkara, A., Challacombe, M., Chen, W., Wong, M.W., Pople, J.A.: Gaussian Development Version, Revision G.03+. Gaussian, Inc., Wallingford
2. OpenMP Application Program Interface, Version 3.0 (May 2008), `http://www.openmp.org/mp-documents/spec30.pdf`
3. Pople, J.A., Raghavachari, K., Schlegel, H.B., et al.: Int. J. Quantum Chem. Quant. Chem. Symp. S13, 225 (1979)
4. Frisch, M.J., Head-Gordon, M., Pople, J.A.: Chem. Phys. 141(2-3), 189 (1990)
5. Osamura, Y., Yamaguchi, Y., Saxe, P., et al.: J. Mol. Struct (Theochem.) 12, 183 (1983)
6. Pulay, P.: J. Chem. Phys. 78(8), 5043 (1983)
7. Boys, S.F.: Proc. Roy. Soc. Ser. A 542 (1950)
8. Roothaan, C.C.J.: Rev. Mod. Phys. 23(2), 69 (1951)
9. Roothaan, C.C.J.: Rev. Mod. Phys. 32(179) (1960)
10. Stephens, P.J., Devlin, F.J., Ashvar, C.S., et al.: Faraday Discuss 99, 103 (1994)
11. Greengard, L., Rokhlin, V.: J. Comp. Phys. 73(2), 325 (1987)
12. Strain, M.C., Scuseria, G.E., Frisch, M.J.: Science 271(5245), 51 (1996)

13. Burant, J.C., Strain, M.C., Scuseria, G.E., et al.: Chem. Phys. Lett. 258, 45 (1996)
14. Stratmann, R.E., Burant, J.C., Scuseria, G.E., et al.: J. Chem. Phys. 106(24), 10175 (1997)
15. Izmaylov, A.F., Scuseria, G., Frisch, M.J.: J. Chem. Phys. 125(104103), 1 (2006)
16. Pulay, P.: Ab Initio Methods in Quantum chemistry - II, p. 241 (1987)
17. Devlin, F.J., Stephens, P.J., Cheeseman, J.R., et al.: J. Phys. Chem. A 101(35), 6322 (1997)
18. Devlin, F.J., Stephens, P.J., Cheeseman, J.R., et al.: J. Phys. Chem. A 101(51), 9912 (1997)
19. Lenoski, D.E., Weber, W.-D.: Scalable Shared-Memory Mulitprocessing. Morgan Kaufmann Publishers, San Francisco (1995)
20. SGI Altix 450 servers, `http://www.sgi.com/products/servers/altix/450/`
21. SGI Histx performance analysis tools, `http://www.sgi.com/products/software/linux/histx.html`
22. Intel® Itanium® 2 Processor Reference Manual for Software Development and Optimization, `http://download.intel.com/design/Itanium2/manuals/25111003.pdf`

Providing Observability for OpenMP 3.0 Applications

Yuan Lin[1] and Oleg Mazurov[2]

[1] Nvidia Corp., Santa Clara, CA 95050, USA
yulin@nvidia.com
[2] Sun Microsystems Inc., Menlo Park, CA 94025, USA
oleg.mazurov@sun.com

Abstract. Providing observability for OpenMP applications is a technically challenging task. Most current tools treat OpenMP applications as native multi-threaded applications. They expose too much implementation detail while failing to present useful information at the OpenMP abstraction level. In this paper, we present a rich data model that captures the runtime behavior of OpenMP applications. By carefully designing interactions between all involved components (compiler, OpenMP runtime, collector, and analyzer), we are able to collect all needed information and keep overall runtime overhead and data volume low.

1 Introduction

For any programming environment, offering observability in the runtime behavior of user applications is equally as important as offering schemes that help to create the application. OpenMP provides a set of high level constructs and APIs that aim to simplify the task of writing high performance and portable parallel applications. The nondeterministic nature of concurrent execution of threads, the program transformations performed by the compilers, and the interactions between user applications and runtime libraries makes program observation (e.g., performance profiling and debugging) more important, and at the same time, more difficult to achieve.

For example, a generic performance analysis tool can only provide rudimentary support for OpenMP performance profiling. The tool may show all native threads used in the process - some may map to OpenMP threads and some may be internal service threads used by the OpenMP runtime library itself. The tool may show native callstacks that barely resemble the caller-callee relationship in user program. The tool may not be able to differentiate the user CPU time used for real work from those used for OpenMP synchronization or caused by OpenMP overhead. In OpenMP, an OpenMP thread executes on behalf of an OpenMP task inside an OpenMP parallel region (implicit or explicit) at any particular moment. If the tool has no knowledge of this context, it cannot present information close to the OpenMP execution model, reducing the help it provides in trouble-shooting OpenMP specific performance problems. To a certain extent, the OpenMP execution environment resembles a virtual machine. We believe, for general OpenMP programmers, observability should be provided at the OpenMP abstraction level, in addition to the machine level which exposes internals of the virtual machine.

This paper makes the following contributions. First, it presents a data model upon which the runtime behavior of an OpenMP program can be constructed for observation. Second, it presents a way to extend the current OpenMP profiling API to support the new OpenMP tasking feature, and a way to integrate vendor specific extensions. Third, it presents the techniques for generating the user callstack that reflects the program logic rather than implementation details. And last, it presents the techniques for efficiently generating, recording, and reconstructing various tree structures in the data model.

The rest of the paper is organized as follows. Section 2 describes our OpenMP data model. Section 3 describes extensions to the OpenMP profiling API to support the data model. Section 4 presents our techniques in getting the user callstack. Section 5 presents our techniques in getting the thread tree and the task tree related information. Section 6 uses a quicksort program to illustrate the information that can be constructed and presented to users using the data collected by our techniques. Section 7 describes related work. Section 8 concludes our paper.

2 The Data Model

In this section, we describe a rich *data model* that captures the runtime behavior of an OpenMP application. Any event of interest that happened during the execution of an OpenMP application will have its OpenMP context described in the data model.

Before diving into the details of the data model, we define the components involved in the tool chain:

- A *compiler* that processes program source code, translates OpenMP constructs, produces executable code functionally equivalent to the original, and provides information that allows mapping of various program objects in the executable code back to the source code (e.g., user and outlined function naming, instruction to source line mapping).
- An *OpenMP runtime library* that provides thread/task management, scheduling and synchronizations.
- A *collector* that is a runtime component interacting with the program and the OpenMP runtime to obtain and to record information about dynamic program behavior. Low overhead is crucial for this component. Thus recorded data may be different from what finally makes the data model.
- An *analyzer* that works either on-the-fly or post-mortem. The analyzer implements the data model by reconstructing it from data recorded by the collector and provides a means for the user to access that data model through a set of displays or tools with various mechanisms to manipulate data, such as filtering and aggregation.

The callstack is a crucial piece of information for observability. We need a callstack that hides the details of outlined functions from a user, contains only stack frames from the user's code and maintains natural caller/callee relationships as specified by the logic of the user's program. A set of collected callstacks can be formed into a tree-like data structure with each path from the root representing a particular callstack. We'll refer to that data structure as the dynamic function call graph or function tree.

OpenMP parallel region information is required to understand the parallel behavior of a program. By assuming that serial program execution is represented by the default parallel region with only one thread in the team, we can assert that any program event occurs in some OpenMP thread (i.e., a member of the current parallel region thread team). With nested parallelism and a parent-child relationship between parallel regions we get a tree-like data structure where we can map any event to a node thus specifying all ancestor OpenMP threads and parallel regions up to the default one, which is the root of the tree. We call that data structure the OpenMP thread tree or parallel region tree.

The OpenMP runtime provides a unique ID for each parallel region instance. To collect and to present the entire dynamic parallel region tree would require an enormous amount of data. Two optimizations appear practical: a sampled parallel region tree; and a representation in which all dynamic IDs are mapped to original OpenMP directives in the source code while preserving the dynamic parent-child relationship. This is similar to the dynamic function call graph.

During their lifetime, OpenMP threads transition through various states defined by the OpenMP runtime, such as working in user code, waiting in a barrier or other synchronization objects and performing a reduction operation. Time spent in different states can be aggregated and presented as metrics for various program objects: functions, threads, parallel regions, etc. An OpenMP thread state is thus another important part of the data model.

OpenMP tasks give another perspective to a program's dynamic behavior. By extending the notion of a task for serial execution, parallel regions, worksharing constructs, etc. (and calling such tasks implicit), we can assert that any program event is associated with some task. Task creation defines a parent-child relationship between tasks, which leads us to another tree-like data structure - the OpenMP task tree. Optimizations similar to those suggested for the parallel region tree are also possible.

The OpenMP data model thus consists of the following pieces of information defined for an arbitrary program event: an OpenMP thread state, a node in the parallel region tree, a node in the task tree, and a user callstack. The actual data model may also contain information that is not OpenMP related, such as a time-stamp, a machine callstack, a system thread ID or a physical CPU ID associated with a particular event of interest. We do not discuss how to record and process such information as the techniques are mature and well-known.

3 OpenMP Profiling API

The original OpenMP profiling API[1] was designed to allow for easy extensions, and the introduction of tasks in OpenMP 3.0[2] created an immediate need for such extensions.

To collect information corresponding to the data model described in the previous section, a new set of nine requests is proposed for the common API:

1. depth of the current task tree;
2. task ID of the n-th ancestor task (0 for the current task);
3. source location of the n-th ancestor task;
4. depth of the current parallel region tree;
5. parallel region ID of the n-th ancestor parallel region (0 for the current one);
6. source location of the n-th ancestor parallel region;
7. OpenMP thread ID of the n-th ancestor thread in the parallel region tree path;
8. size of the thread team of the n-th ancestor parallel region;
9. OpenMP state of the calling thread.

Notice that although some of the above information (such as the size of the thread team) is available through standard OpenMP API calls (such as `omp_get_team_size()`), the collector runtime should use the profiling API[1] because it guarantees the above nine pieces of information are provided consistently and atomically in one API call.

Although not used in the work presented in this paper, a set of new events is proposed to cover tasks in OpenMP 3.0: a new task created, task execution begins, task execution suspended, task execution resumed, task execution ends (Appendix A).

Recognizing that some interactions between the collector and the OpenMP runtime can be very specific to a particular implementation, we are suggesting a simple way to add vendor specific requests and events, which would not interfere with possible future extensions of the common API. A vendor willing to implement its own set of extensions should reserve one request number in the common API to avoid possible collision of similar requests from other vendors. This request is issued during the rendezvous to check if that vendor's extensions are supported by the OpenMP runtime and if so, to enable them. All actual extended requests and events are assigned negative values, which will never be used by the common API, and are put in a separate include file. This scheme assumes that no two sets of extensions can be enabled at the same time but it allows both the OpenMP runtime and the collector to support more than one vendor extension. Thus, there is no problem with possible overlap of values or names of actual extension requests and events defined by different vendors.

4 Collecting User Call Stack Information

4.1 The Challenges

OpenMP 3.0 introduces a new tasking feature[1] which makes it easier to write more efficient parallel applications that contain nested parallelism or dynamically

[1] For conciseness, we use the OpenMP task construct to illustrate the challenges and our solution, as it is the most difficult construct to deal with. Similar techniques can be applied to other OpenMP constructs.

generated concurrent jobs. To allow for concurrency, the execution of a task can be deferred and the thread that executes a task may be different from the thread that creates the task. In Fig. 4.1 (a), function `goo()` may be executed by thread 1 while function `bar()` may be executed by thread 2. And function `bar()` may be executed after function `foo()` has returned.

Most OpenMP implementations use the *outlining* technique that generates an *outlined* function that corresponds to the body of many OpenMP constructs and uses a runtime library to schedule the execution of outlined functions among OpenMP threads. Fig. 4.1 (b) illustrates the transformed code. Fig. 4.1 (c) illustrates the interaction between the compiler generated code and the OpenMP runtime library. Notice that function `foo()` now calls an entry point `_mt_TaskFunction_()` in the OpenMP runtime which may asynchronously executes the encountered task on another thread.

Getting the user callstack is not straightforward. In Fig. 4.1, when we inspect the native call stack while the program is executing `bar()`, the native callstack will be very different from the native callstack in code that does not have the OpenMP construct. Fig. 4.1 (d) illustrates the differences. First, the native callstack has frames that are from the OpenMP runtime library. Second, the outlined function is called by a slave thread in a dispatching function inside the OpenMP runtime library. The frames from the root down to the outlined functions are all from the runtime library. Last but not least, function `foo()` may have returned. None of the native callstacks in any of the threads show where the task associated with `__foo_task1()` comes from. All these complications are implementation details that users usually do not care about, have no knowledge of, and are often confused by. To make things worse, the internal implementation scheme may change from one version of implementation to another.

4.2 Scheme Overview

At any moment in an OpenMP application, a thread is executing some OpenMP task if it is not idle waiting for work. Therefore, the user callstack (UC) for any event is made of two pieces: *task spawn user callstack (TSUC)* and *local segment (LS)*.

$$UC = TSUC + LS$$

The $TSUC$ is the user callstack for the spawn event of the current task. The LS is the callstack corresponding to the execution of the outlined task function.

Let's assume for the moment that we know how to get the local segment, then the basic scheme of constructing the user callstacks becomes quite straightforward. When a task is spawned, we get the user callstack corresponding to the spawn event and store it together with the data structure for the task itself. This user callstack, excluding the PC that calls `_mt_TaskFunction_`, is the TSUC for any subsequent event that happened during the execution of the task. Each thread maintains a record of the current task it is executing. When an event happens, we can find the TSUC of the current task by querying the task data structure. We concatenate it with the local segment and get the user callstack.

```
kar()                  | kar()
{                      | {
   foo();              |     foo();
}                      | }
                       |
foo()                  | foo()
{                      | {
    goo();             |     goo();
    #pragma omp task   |     taskinfo = ...
    {                  |     _mt_TaskFunction_(taskinfo, __foo_task1, ...);
       bar();          | }
    }                  |
}                      | __foo_task1(char *arg)
                       | {
                       |     bar();
                       | }
                       |
bar()                  | bar()
{                      | {
   statement 1;        |    statement 1;
}                      | }

    (a)                         (b)
```

```
         User's Code                  .          OpenMP runtime library
   +--------------------------------------------------------------------------
   |  foo()                           .
 T |  {                               .
 h |     goo();                       .
 r |     taskinfo = ...               .
 e |     _mt_TaskFunction_(taskinfo, ----->  _mt_TaskFunction_(taskinfo,
 a |               __foo_task1, ...);  .                    mfunc_entry,...)
 d |  }                               .     {    ...
   |                                  .          stored_entry = mfunc_entry;
 1 |                                  .          ...
   |                                  .     }
.............................................................................
   |                                  .
 T |                                  .      dispatcher ()
 h |                                  .      {
 r |                                  .           ...
 e |   __foo_task1(char *arg)        <-----      (*stored_entry)(...);
 a |   {                              .           ...
 d |       bar();                     .       }
   |   }                              .
 2 |                                  .
   |                                  .

                              (c)
```

```
Native Callstack                              User Callstack
=================                             ==============
_lwp_start()                                  main()
<frames in the OpenMP runtime library>        kar()
dispatcher()                                  foo()
__foo_task1()                                 bar()
bar()
                                  (d)
```

Fig. 1. (a) original code; (b) compiler transformed code; (c) interaction between compiled code and OpenMP runtime library; (d) native callstack vs user callstack

Notice that the construction of user callstacks is conceptually recursive, and the *TSUC* is always empty for the outer-most task.

In the rest of this section, we will discuss how to get the local segment, and will present a method to get the TSUC more efficiently since computing the user callstack for each task spawn event can be very expensive.

4.3 Getting the Local Segment

The local segment can be constructed by walking up the native callstack when an event happens. Stopping at the first frame from the OpenMP runtime library does not work, because (a) the program may be executing an OpenMP user routine (e.g. `omp_set_num_threads()`); (b) the program may be inside some library that the OpenMP runtime library calls (e.g. `memcpy()` in the standard C library). The key is to tell whether the program is inside an OpenMP user routine, and whether the program is inside an outlined function.

The OpenMP runtime maintains an *in_omp_user_api_state* flag, which is set to 1 whenever the program enters an OpenMP user routine, and is reset to 0 when the program leaves the OpenMP user routine. The OpenMP runtime also maintains a *boundary_stack_pointer*. When the OpenMP runtime library is about to call an outlined function, it records, in the *boundary_stack_pointer*, a stack location in a frame in the call-chain within the OpenMP runtime that will eventually lead to the outlined function. The OpenMP runtime reports the *in_omp_user_api_state* and the *boundary_stack_pointer* to the collector upon request. Section 4.6 describes how the collector uses the two values.

4.4 Getting the Task Spawn User Callstack

The task spawn user callstack (TSUC) is essentially the user callstack (excluding the PC that calls `_mt_TaskFunction_()` (see Fig. fig:exe)) when the task is spawned. Since any task in an invocation of a function will have the same TSUC, we can get the TSUC at the entry of the function instead of computing it every time a task is spawned. This reduces the overhead when multiple tasks are created in one function call, for example inside a loop.

4.5 Pragma PC

In the above description, we assume that, when a profiling event occurs, the thread is either executing some user code or some OpenMP user API calls. However, the thread may also be executing some OpenMP runtime library code (e.g., a thread is enqueuing, dequeuing, or stealing a task). At these moments, the local segment is empty. We are not able to get the leaf PC for the user callstack from the empty segment. The leaf PC that can be used naturally in the user callstack at these moment would be the call instruction to `_mt_TaskFunction_()`. So, we need to get this PC, which we call *pragma_pc*, and report it to the collector, which will use this PC when it finds the local segment is empty.

During the lifetime of a task region, its TSUC and *pragma_pc* will remain constant, while its *boundary_stack_pointer* is set only when the corresponding outlined function is about to be executed.

4.6 Collector Runtime Behavior

We described an idea of storing the task spawn user callstack in the data structure representing a task above in Section 4.2. In our case, a callstack is an array of unprocessed virtual addresses obtained by a stack walking routine implemented in the collector. However, to optimize memory and disk use, the collector implements a mapping scheme using a simple hashing technique. An array of addresses is mapped into a unique 64-bit identifier (UID). The UID can be passed around during program execution, recorded along with other data at an arbitrary time. The mapping scheme guarantees the array of addresses can be reconstructed from the UID during data processing later. The collector records all such mappings on the disk and keeps track of mappings already recorded to reduce data volume. Although unlikely in usual practice, hash collision is still a possibility. Some hash collisions can be detected and reported from different mapping records with the same UID at analysis time. The probability of an undetected hash collision and its cost are considered negligible for the task of statistical performance profiling.

It is trivial to extend to the hashing algorithm so that a new UID can be computed from some previous UID and a segment of addresses. This is essentially the same as appending the segment of address to the array of addresses represented by the previous UID. Therefore, the task spawn user callstack can be represented using a 64-bit UID, which we call *mfunc_start_context_id*. The *mfunc_start_context_id* is obtained by the OpenMP runtime from the collector, stored in a task data structure, and reported back to the collector whenever requested.

We extended the OpenMP profiling API with a new mechanism that allows the OpenMP runtime to issue requests that are carried out by the collector. During the initial rendezvous the collector registers a helper function as a specific event callback. This helper function is called by the OpenMP runtime to obtain a UID for the current user callstack at moments corresponding to task spawning as described above.

When the collector wants to get a user callstack for some program event, it issues a specific OpenMP profiling API request to get the current context, which includes *mfunc_start_context_id*, *boundary_stack_pointer*, *pragma_pc*, and *in_omp_user_api_state*. The collector uses *boundary_stack_pointer*, *pragma_pc*, and *in_omp_user_api_state* to construct the local stack segment. It walks the stack up to the frame pointed to by *boundary_stack_pointer* and collects PC addresses from all stack frames. The collector then checks the collected addresses in the reverse order and if it finds an address from the OpenMP runtime it cuts off the entire tail, possibly leaving the entry address if *in_omp_user_api_state* is set. If the resulting segment is empty, the collector uses *pragma_pc* for the local segment. The collector then computes a UID for the entire user callstack from *mfunc_start_context_id* and the local segment. The computed UID is recorded along with other profiling data for the event.

Because the overall scheme of maintaining user callstacks for OpenMP tasks includes specific mechanisms and interactions, such as UID computation and the helper mechanism, the request to get the current context is made part of the vendor specific extension to the OpenMP profiling API (Appendix A).

5 Collecting Parallel Region Tree and Task Tree Information

5.1 OpenMP Run Time Part

At any moment, the OpenMP runtime should be able to report, upon query by the collector, the nine pieces of information as described in Section 3. In order to do that, the OpenMP runtime needs to maintain a dynamic tree path during the execution. This is straightforward to implement.

5.2 Collector Part

Instead of collecting and recording information for the entire path in the parallel region tree for each event, the collector tries to reduce overhead and data volume by maintaining the current parallel region ID for each thread in thread local storage (TLS) and recording all necessary information only when it changes. As with callstack UIDs, the collector keeps track of already recorded parallel region IDs. At an event, the collector asks about the current parallel region ID. If the ID is not different from the ID currently stored in TLS, the collector does nothing. Otherwise it records a “thread enters a parallel region” event along with the time-stamp and starts checking if it also needs to record all information about the new parallel region and its ancestors. As multiple threads may almost simultaneously enter a new parallel region, usually only one thread records all information about that parallel region. No synchronization is used between threads to keep track of the recorded status of a parallel region as we only get multiple identical records in the worst case.

At analysis time, we can map any event that is recorded with a time-stamp and a thread ID to an interval determined by “thread enters a parallel region” events, thus obtaining the corresponding parallel region ID. It’s guaranteed by the scheme described above that all information about that parallel region and all its ancestors has also been recorded.

The collector uses the same scheme to record task tree information as for recording parallel region tree information.

6 OpenMP Profiling API Examples

We use a quick sort implementation to illustrate how the ideas described above can be presented to the user by a performance analysis tool. All screenshots are obtained from a prototype based on the Sun Studio^{TM}Performance Analyzer.

A parallel version of the algorithm using OpenMP is shown in Fig. 6 and the code is pretty straightforward.

```
quick_sort(int lt, int rt, float *data)
{
    if ( (rt-lt) < LOW_LIMIT ) {
        serial_quick_sort( lt, rt, data );
    }
    else {
        int md = partition( lt, rt, data );
        #pragma omp task
        quick_sort( lt, md-1, data );
        #pragma omp task
        quick_sort( md+1, rt, data );
    }
}

main(int argc, char* argv[])
{
    int n; float *data;
...
    #pragma omp parallel
    {
        #pragma omp single nowait
        quick_sort( 0, n-1, data );
    }
...
}
```

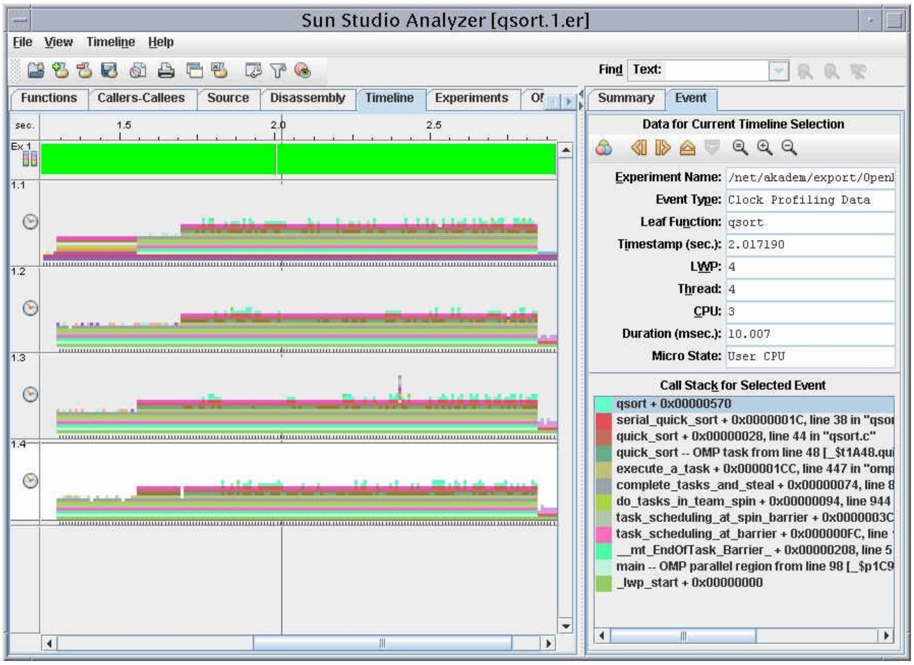
Fig. 2. Parallel quick sort algorithm using OpenMP

An OpenMP unaware tool that is capable of collecting only actual machine callstacks will not show any recursion, because the implementation of OpenMP tasks essentially turns the recursive execution into a work-list based execution. The compiler transformed function of `quick_sort()` contains the initial condition checking, either calls the serial sort function or partitions the specified part of the array and creates two more tasks for sorting both parts of the partition. Instead of recursively calling `quick_sort()`, the program recursively creates tasks. When a task is picked up for execution by a thread, it bears no trace of where it was created and thus the machine callstack has practically the same depth no matter what the logical depth of a particular task is. This behavior can be easily observed in the machine view (Fig. 3 (a)) in Analyzer's Timeline display.

Here, the horizontal axis represents time, and each horizontal bar represents a thread with all collected events shown with their callstacks colored by frame. Selected event details, including the callstack, are shown in the right panel.

In the user view (Fig. 3 (b)), where the logical structure of dynamic program execution is reconstructed, one can see that the recursion pattern with a fluctuating callstack depth is restored.

Knowing the current task ID for each event, we can map it to the original OpenMP construct and compute OpenMP metrics aggregated for those constructs over all events. Two OpenMP metrics, OpenMP Work and OpenMP Wait, are computed based on the OpenMP state recorded for every event. A sorted list of all OpenMP task constructs along with their metrics is presented in the OpenMP tasks display (Fig. 4).



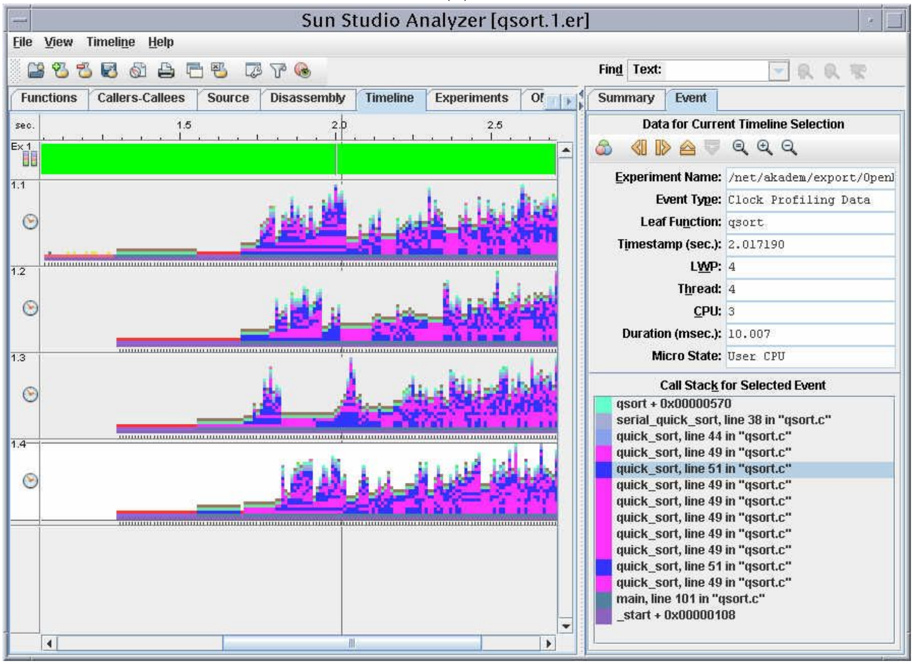
Data for Current Timeline Selection

Experiment Name: /net/akadem/export/OpenI
Event Type: Clock Profiling Data
Leaf Function: qsort
Timestamp (sec.): 2.017190
LWP: 4
Thread: 4
CPU: 3
Duration (msec.): 10.007
Micro State: User CPU

Call Stack for Selected Event

qsort + 0x00000570
serial_quick_sort + 0x0000001C, line 38 in "qsor
quick_sort + 0x00000028, line 44 in "qsort.c"
quick_sort -- OMP task from line 48 [_$t1A48.qui
execute_a_task + 0x000001CC, line 447 in "omp
complete_tasks_and_steal + 0x00000074, line 8
do_tasks_in_team_spin + 0x00000094, line 944
task_scheduling_at_spin_barrier + 0x0000003C
task_scheduling_at_barrier + 0x000000FC, line
__mt_EndOfTask_Barrier_ + 0x00000208, line 5
main -- OMP parallel region from line 98 [_$p1C9
_lwp_start + 0x00000000

(a)



Data for Current Timeline Selection

Experiment Name: /net/akadem/export/OpenI
Event Type: Clock Profiling Data
Leaf Function: qsort
Timestamp (sec.): 2.017190
LWP: 4
Thread: 4
CPU: 3
Duration (msec.): 10.007
Micro State: User CPU

Call Stack for Selected Event

qsort + 0x00000570
serial_quick_sort, line 38 in "qsort.c"
quick_sort, line 44 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 51 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 49 in "qsort.c"
quick_sort, line 51 in "qsort.c"
quick_sort, line 49 in "qsort.c"
main, line 101 in "qsort.c"
_start + 0x00000108

(b)

Fig. 3. (a) Machine View; (b) User View

```
OMP Work  OMP Wait   Name
 sec.      sec.
----------------------------------------------------------------------
6.254     1.861      <Total>
2.512     0.010      OpenMP task from quick_sort, line 48 in "qsort.c"
2.342     0.         OpenMP task from quick_sort, line 50 in "qsort.c"
1.141     0.771      OpenMP task 0
0.260     1.081      OpenMP task from main, line 98 in "qsort.c"
```

Fig. 4. OpenMP tasks display

A similar display is provided for all parallel regions. Again, dynamic parallel region IDs are mapped to source and all metrics are computed for the corresponding OpenMP constructs.

7 Related Work

The need for a user level, implementation independent representation of OpenMP program behavior that is consistent with the OpenMP programming model is generally desired and was, in particular, stated in [3] for OpenMP debugging. [3] also emphasized the importance of more detailed views that expose underlying implementation specifics for sophisticated users.

A work towards an open source implementation of the OpenMP profiling API has been reported in [4].

The problem of uniquely identifying OpenMP threads with nested OpenMP parallelism has been approached in [5], where a suggestion, similar to ours, for extension of the standard OpenMP runtime API was made.

User call stacks can also be obtained by tracing function entry and exit events and by maintaining a data structure that allows reconstruction of user call stacks at run time. ompP[6] uses a similar approach to track OpenMP parallel region entry and exit events. It is not clear whether this technique can be extended to deal with tasks without imposing significant overhead, because there usually are significantly more tasks than parallel regions. It is also unclear how this technique would handle untied tasks. Yet another major challenge is dealing with *survived tasks* - a task whose ancestor tasks have finished before the task starts executing.

A general method for efficiently collecting logical call path profiles in multithreaded applications and its implementation for Cilk are described in [7]. The method relies on the availability in the runtime of all pieces of information necessary for logical call path reconstruction at an arbitrary sample point. While that method can certainly cope with work-stealing, as implemented in both Cilk and OpenMP, it's not obvious how it would grapple with survived tasks, which are prohibited by design in Cilk but are allowed in OpenMP.

8 Conclusion

For OpenMP runtime observation tools, such as a debugger and a performance profiling tool, the user model should be intuitive and close to program logic, and should be presented in terms of high level language constructs used in the program. In this paper, we present a rich data model, which comprises a function tree, a parallel region tree and a task tree, that captures the OpenMP specific runtime behavior. We describe a set of methods that efficiently collect the data for the data model. This work demonstrates that providing high level observability to OpenMP programming and runtime systems, though challenging, is achievable.

Acknowledgements

The authors would like to thank Martin Itzkowitz, Eric Duncan, and Nawal Copty for reviewing the paper. The authors would also like to thank the anonymous reviewers for their many helpful comments and suggestions.

References

1. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: White Paper: An OpenMP Runtime API for Profiling. Tech. Rep., Sun Microsystems, Inc. (2007)
2. OpenMP Architecture Review Board. OpenMP application program interface, version 3.0 (2008), `http://www.openmp.org/mp-documents/spec30.pdf`
3. Cownie, J., DelSignore Jr., J., de Supinski, B., Warren, K.: DMPL: An OpenMP DLL Debugging Interface. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 137–146. Springer, Heidelberg (2003)
4. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Gopalkrishnan, P., Tafti, D.: Towards an Implementation of the OpenMP Collector API. In: Proceedings of the International Conference ParCo (2007)
5. Morris, A., Malony, A., Shende, S.: Supporting Nested OpenMP Parallelism in the TAU Performance System. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 279–288. Springer, Heidelberg (2008)
6. Fuerlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 15–23. Springer, Heidelberg (2008)
7. Nathan, R.: Tallent and John Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, New York (2009)

A OpenMP Profiling API Extensions

```
/* New requests */
typedef enum {
        ...
        OMP_REQ_TASK_NLVLS,     /* depth of the current task tree */
        OMP_REQ_TASK_IDN,       /* task ID of the n-th ancestor task */
        OMP_REQ_TASK_SRCN,      /* source location of the n-th ancestor task */
        OMP_REQ_PREG_NLVLS,     /* depth of the current parallel region tree */
        OMP_REQ_PREG_IDN,       /* ID of the n-th ancestor parallel region */
        OMP_REQ_PREG_SRCN,      /* source location of the n-th ancestor parallel region */
        OMP_REQ_PREG_THRIDN,    /* thread ID of the n-th ancestor thread */
        OMP_REQ_PREG_TMSZN,     /* thread team size of the n-th ancestor parallel region */
        OMP_REQ_CREATED_TASK,   /* task ID of a newly created task */
} OMP_COLLECTORAPI_REQUEST;


/* New events */
typedef enum {
        ...
        OMP_EVENT_CREATE_TSK,   /* a new task created */
        OMP_EVENT_BEGIN_TSK,    /* task execution begins */
        OMP_EVENT_SUSPEND_TSK,  /* task execution suspended */
        OMP_EVENT_RESUME_TSK,   /* task execution resumed */
        OMP_EVENT_END_TSK       /* task execution ends */
} OMP_COLLECTORAPI_EVENT;

/* New OpenMP thread state */
typedef enum {
        ...
        THR_TSKWT_STATE,        /* waiting in taskwait */
} OMP_COLLECTOR_API_THR_STATE;

/* Reserve request ID for Sun specific extensions */
#define OMP_REQ_SUNEXTENSION  ((OMP_COLLECTORAPI_REQUEST)0x4A415641)

/* Sun specific extensions (from a separate include file) */

#define OMPX_REQ_CONTEXT      ((OMP_COLLECTORAPI_REQUEST)-1)

struct OMPX_request_context {
    int                         size;       /* entry length   */
    OMP_COLLECTORAPI_REQUEST    reqn;       /* request number */
    OMP_COLLECTORAPI_EC         errc;       /* error code     */
    int                         rtsz;       /* return size    */
    uint64_t                    mfunc_start_context_id;
    void                        *boundary_stack_pointer;
    void                        *pragma_pc;
    int                         pragma_pc_state;
    int                         in_omp_user_api;
};

#define OMPX_REGISTER_HELPER  ((OMP_COLLECTORAPI_EVENT)-1)

#define OMPX_HLP_UCTX         ((OMP_COLLECTORAPI_REQUEST)-2)

struct OMPX_helper_uctx {
    int                         size;   /* entry length   */
    OMP_COLLECTORAPI_REQUEST    reqn;   /* request number */
    OMP_COLLECTORAPI_EC         errc;   /* error code     */
    int                         rtsz;   /* return size    */
    void                        *starting_stack_pointer;
    void                        *boundary_stack_pointer;
    uint64_t                    mfunc_start_context_id;
    void                        *pragma_pc;
    uint64_t                    new_context_id; /* return result */
};
```

A Microbenchmark Suite for Mixed-Mode OpenMP/MPI

J. Mark Bull, James P. Enright, and Nadia Ameer

EPCC, The King's Buildings, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
m.bull@epcc.ed.ac.uk

Abstract. With the current prevalence of multi-core processors in HPC architectures, mixed-mode programming, using both MPI and OpenMP in the same application, is becoming increasingly important. However, no low-level synthetic benchmarks exist to test the performance of this programming model. We have designed and implemented a set of microbenchmarks for mixed-mode programming, including both point-to-point and collective communication patterns. These microbenchmarks have been run on a number of current HPC architectures: the results show some interesting performance differences between the architectures and highlight some possible inefficiencies in the implementation of MPI on multi-core systems.

1 Introduction

With the advent of multi-core processors, and the associated diminishing rate of increase in processor clock speed, almost all current high performance computing systems now contain nodes which consist of shared memory multiprocessors. Large numbers of such nodes can be connected together with a high-bandwidth, low-latency network to form a scalable distributed memory system.

To program such systems, by far the most popular programming model is message-passing, using the MPI [4] library. MPI programs can execute on machines with shared memory nodes in a straightforward way by running one MPI process per core on each node. In this case, message-passing between processes on a node is normally implemented via shared memory, but this is not visible to the programmer. However, it is also possible to run fewer MPI processes than cores on each node, and make use of the additional cores by using a multi-threaded programming model. This is most frequently done using the OpenMP [5] API, but can also be accomplished via a lower-level thread library interface such as Posix threads. This programming style is termed *mixed-mode* or *hybrid* (we prefer the former term as the latter is somewhat overloaded in the HPC literature).

Several studies (for example [7],[10]) have shown that, in certain circumstances, mixed-mode programs can perform better than (or consume less memory than) the equivalent program using MPI only. Such advantages may outweigh the potential additional software complexity and possible loss of portability of the mixed-mode version. A number of commonly used HPC applications, such as CPMD [2] and ECMWF's Integrated Forecast System (IFS) [9] successfully exploit mixed-mode programming. With the principal performance gains in HPC architectures likely to come, in the near future at least, primarily from increasing the number of cores per chip, mixed-mode programming seems likely to assume a more important role, as it may allow applications to scale better on such systems than pure MPI.

Microbenchmarks (low-level synthetic benchmarks testing the performance of basic operations) exist for both MPI [3], [8] and OpenMP [1]. However, these microbenchmarks cannot on their own give sufficient information about the performance of mixed-mode programs, as there will, in general, be interactions between the MPI and OpenMP layers. The Sphinx benchmark suite from LLNL [11] contains a small number of OpenMP/MPI microbenchmarks, which measure the performance of mixed-mode barriers and reductions, and assess the ability to overlap threaded computation with MPI non-blocking communication.

To fill this gap, we have designed and implemented a suite of microbenchmarks for mixed-mode OpenMP/MPI programming. The utility of such a suite is demonstrated by the results presented in [6], which demonstrate how the available communication bandwidth between nodes can depend on the mix of MPI processes and OpenMP threads employed.

In Section 2, we describe the contents of the suite and the rationale for its construction. In Section 3, we present selected results from running the microbenchmarks on a number of current HPC architectures, and demonstrate the interesting features thus illuminated. Finally, Section 4 presents our conclusions and possibilities for future work.

2 Benchmark Design and Implementation

The basic design concept of the mixed-mode microbenchmarks is to provide mixed-mode analogues for (a subset of) the typical operations found in MPI microbenchmark suites, for both point-to-point and collective communications. There are two main considerations which have driven the design of the benchmark suite. Firstly, we wish to adequately capture the cost of the inter-thread communication and synchronisation which may occur in mixed-mode programs if not all threads participate in the inter-node (MPI) communication. To do this, we measure not only the cost of the MPI library calls themselves, but also the (possibly multi-threaded) writing of send buffers, and reading of receive buffers. The second consideration is that we wish to be able to directly and easily compare the performance of the same communication patterns when we hold the total number of cores constant, but vary the number of MPI processes and OpenMP threads (such that the product of these two values equals the number of cores). This is achieved by the appropriate choices of data buffer sizes and MPI message lengths.

The benchmarks are implemented in Fortran90. We may produce a C version in the future, but we expect that there would be little dependence on the base language, as most of the performance characteristics are dictated by the hardware and by the MPI and OpenMP libraries.

2.1 Point-to-Point Operations

In the mixed-mode microbenchmark suite we measure the performance of three point-to-point communication operations: PingPong, PingPing and HaloExchange. Each of these operations is implemented in each of three different ways:

1. **Master-only:** MPI communication takes place in the master thread, outside of parallel regions.
2. **Funnelled:** MPI communication takes place in the master thread, inside parallel regions.
3. **Multiple:** MPI communication takes place concurrently in all threads inside parallel regions.

To illustrate this, Figures 1–3 show pseudocode representations of these three forms of the PingPong benchmark. The PingPing benchmark differs from PingPong in that messages are exchanged in both directions between the two processes concurrently. For both the PingPong and PingPing benchmarks, the user can specify the two MPI ranks which participate: this is intended to permit the measurement of both intra-node and inter-node MPI communication, by specifying two MPI ranks which will execute either on the same, or on different nodes. The benchmark reports which of these was the case by comparing the results of `MPI_GET_PROCESSOR_NAME` on the two participating process. For the HaloExchange benchmark all MPI processes participate. The processes are arranged in a ring and each process exchanges messages with its two neighbouring processes. In the point-to-point benchmarks, the data sizes specified by the user correspond to the total number of 4-byte words sent between pairs of MPI processes.

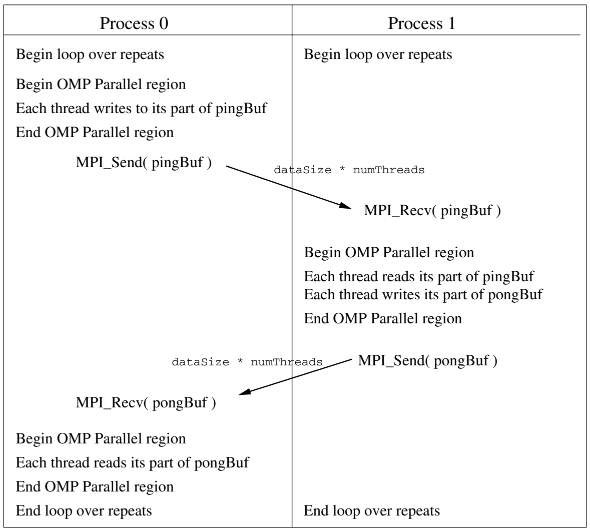


Fig. 1. Pseudocode for Master-only PingPong benchmark

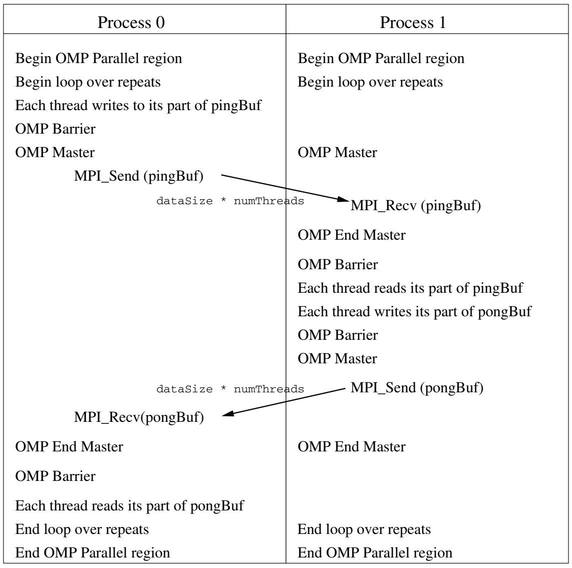


Fig. 2. Pseudocode for Funnelled PingPong benchmark

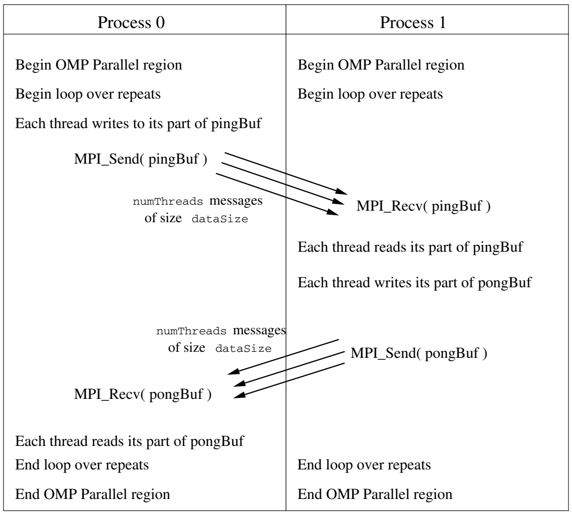

Process 0
Process 1
Begin OMP Parallel region
Begin loop over repeats
Each thread writes to its part of pingBuf
MPI_Send(pingBuf)
numThreads messages of size dataSize
MPI_Recv(pingBuf)
Each thread reads its part of pingBuf
Each thread writes its part of pongBuf
numThreads messages of size dataSize
MPI_Send(pongBuf)
MPI_Recv(pongBuf)
Each thread reads its part of pongBuf
End loop over repeats
End OMP Parallel region

Fig. 3. Pseudocode for Multiple PingPong benchmark

2.2 Collective Operations

The microbenchmark suite contains measurements for mixed-mode analogues of the following operations: Barrier, Reduce, AllReduce, Gather, Scatter and

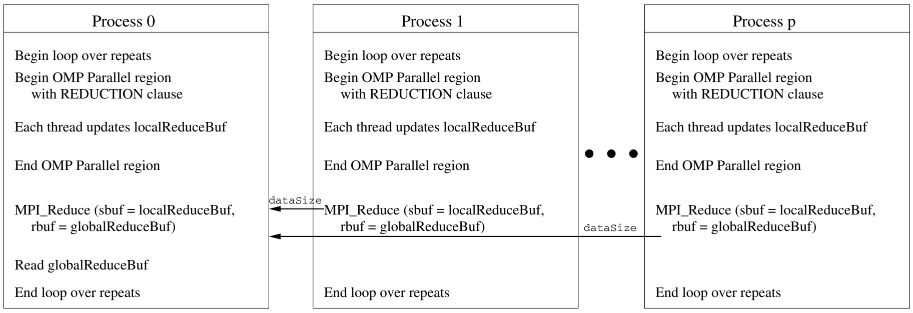

Process 0
Begin loop over repeats
Begin OMP Parallel region with REDUCTION clause
Each thread updates localReduceBuf
End OMP Parallel region
MPI_Reduce (sbuf = localReduceBuf, rbuf = globalReduceBuf)
Read globalReduceBuf
End loop over repeats
Process 1
Begin loop over repeats
Begin OMP Parallel region with REDUCTION clause
Each thread updates localReduceBuf
End OMP Parallel region
MPI_Reduce (sbuf = localReduceBuf, rbuf = globalReduceBuf)
End loop over repeats
Process p
Begin loop over repeats
Begin OMP Parallel region with REDUCTION clause
Each thread updates localReduceBuf
End OMP Parallel region
MPI_Reduce (sbuf = localReduceBuf, rbuf = globalReduceBuf)
End loop over repeats
dataSize
dataSize

Fig. 4. Pseudocode for Reduce benchmark

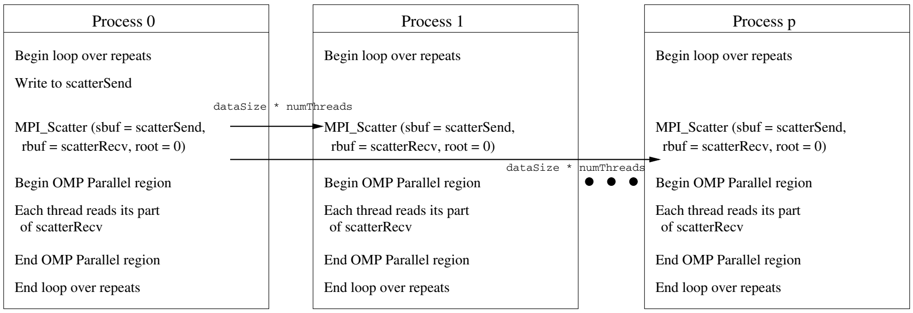

Process 0
Begin loop over repeats
Write to scatterSend
MPI_Scatter (sbuf = scatterSend, rbuf = scatterRecv, root = 0)
Begin OMP Parallel region
Each thread reads its part of scatterRecv
End OMP Parallel region
End loop over repeats
Process 1
Begin loop over repeats
MPI_Scatter (sbuf = scatterSend, rbuf = scatterRecv, root = 0)
Begin OMP Parallel region
Each thread reads its part of scatterRecv
End OMP Parallel region
End loop over repeats
Process p
Begin loop over repeats
MPI_Scatter (sbuf = scatterSend, rbuf = scatterRecv, root = 0)
Begin OMP Parallel region
Each thread reads its part of scatterRecv
End OMP Parallel region
End loop over repeats
dataSize * numThreads
dataSize * numThreads

Fig. 5. Pseudocode for Scatter benchmark

AlltoAll. Figures 4 and 5 show pseudocode representations of the Reduce and Scatter benchmarks respectively.

The other collective benchmarks are constructed in an analogous fashion. The total amount of data involved is proportional to both the number of OpenMP threads and the number of MPI processes. This means that experiments can easily be conducted where the total amount of compute resource, and the product of the number of threads and the number of processes is fixed, but the number of processes and number of threads per process is varied. The benchmarks are constructed so that when this is done, the patterns and quantity of data movement are preserved. (Note that for the Barrier benchmark, no data is involved).

2.3 Benchmark Control

The user is able to set some control parameters for the benchmark suite:

- A list of the benchmarks to be run.
- The minimum and maximum data sizes to be run. The data size starts at the minimum size and is successively doubled until the maximum is reached.
- A target execution time. If the execution time for a given data size is less than this value, it is rejected, and the test is re-run with twice the number

of repetitions. If the execution time is more than twice the target time, it is accepted, and the initial number of repetitions for the subsequent data size is set to half its current value. This process is intended to keep the execution time for each test approximately constant, regardless of the benchmark or the data size used. Before each timed test, two repetitions are run as a warm-up.
- The MPI process IDs to be used for the PingPong and PingPing benchmarks. Negative values are permitted: in this case the value is added to the total number of processes to give a valid ID.

At present, the number of MPI processes and OpenMP threads are controlled by the way the benchmark is executed (i.e. by the `mpirun` command or equivalent, and the value of the `OMP_NUM_THREADS` environment variable) and they are fixed for that run. We considered trying to vary the number of process and threads within a run, but the complexity of the programming and the possible lack of control over idle threads mean we have not yet done so.

2.4 Other Issues

Each benchmark has a validation test, which is run on the warm-up repetitions. For each benchmark and data size a Pass or Fail is reported.

The benchmark reports the value returned by `MPI_INIT_THREAD`, and issues a warning if the level of support is not adequate for the benchmark. The Multiple versions of point-to-point benchmarks require `MPI_THREAD_MULTIPLE`, while all other benchmarks require `MPI_THREAD_FUNNELED`. We have found that the value returned is a poor indicator of whether the validation test will succeed. We have encountered one implementation of MPI which returns `MPI_THREAD_SINGLE` but runs the benchmarks requiring `MPI_THREAD_FUNNELED` successfully, and another implementation which returns `MPI_THREAD_MULTIPLE`, but fails to run the benchmarks requiring this value.

3 Benchmark Results

3.1 Hardware

We have run the benchmark suite on four different platforms:

- **IBM eServer 575 Power5 cluster**. Each node contains 8 1.6GHz dual-core processors and 32GB of memory, and the nodes are connected with IBM's High Performance Switch (HPS) with a total of four links from each node to the network. The system was running Version 10.1 of the IBM xlf90 Fortran compiler and Version 4.3 of IBM Parallel Operating Environment. Our experiments used 4 nodes (64 cores).
- **IBM eServer 575 Power6 cluster**. Each node contains 16 4.7 GHz dual-core processors and 128 or 256GB of memory. The nodes are connected through an Infiniband network with four links from each node to the network.

The system was running Version 12.1 of the IBM xlf90 Fortran compiler and Version 4.3 of IBM Parallel Operating Environment. Our experiments used 4 nodes (128 cores).

- **IBM BlueGene/P**. Each node has four 850MHz Power450 cores and 2GB of memory. There are three networks connecting the compute nodes of the BlueGene/P, a 3D torus network and two tree networks (one used for collective communication, the other for barrier synchronisation). The system was running Version 11.1 of the IBM xlf90 Fortran compiler and BlueGene Driver Version 1.0 Release 3.0. Our experiments used 16 nodes (64 cores).
- **Cray XT4**. Each node contains a quad-core 2.3 GHz AMD Opteron processor and 8 or 16GB of main memory. The network is a Cray SeaStar 3D torus. The system was running Version 7.2.4 of the PGI pgf90 compiler and Version 3.0.2 of the Cray Message Passing Toolkit. Our experiments used 16 nodes (64 cores).

In all cases we fully populated the nodes, so the product of the number of MPI process per node and the number of OpenMP threads per process always equals the number of cores per node.

3.2 Results

We do not have space here to show the results of all the benchmarks on all the platforms, so we have selected some of the more interesting results for presentation.

Figures 6 and 7 show the results of running the Master-only version of the PingPong benchmark on the IBM Power 5 cluster and BlueGene/P system respectively. The execution times are normalised to the execution time with

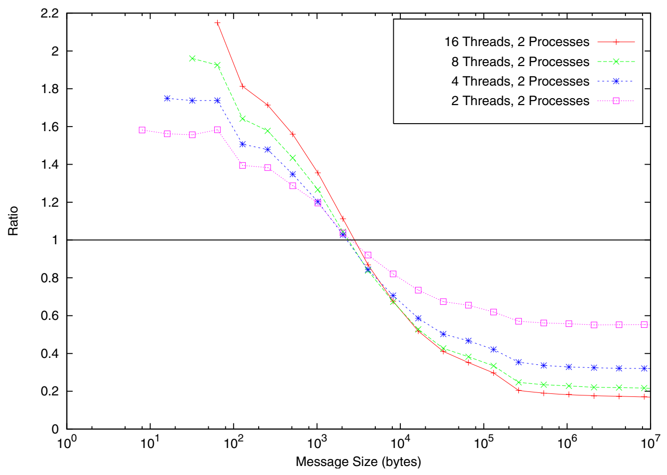


Fig. 6. IBM Power 5: Ratio of time in Master-only PingPong benchmark to one thread per process

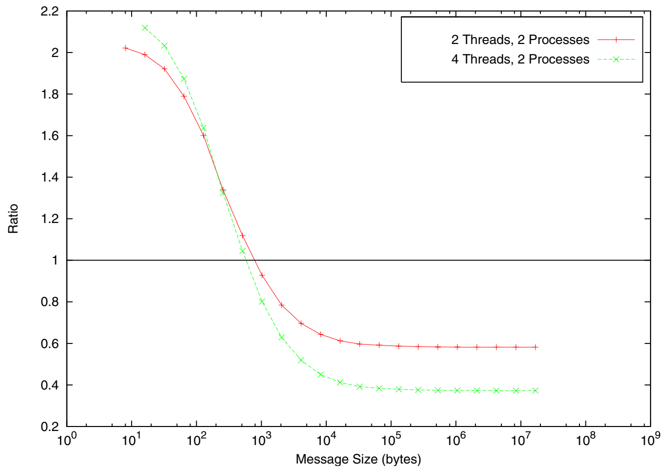


Fig. 7. IBM BlueGene/P: Ratio of time in Master-only PingPong benchmark to one thread per process

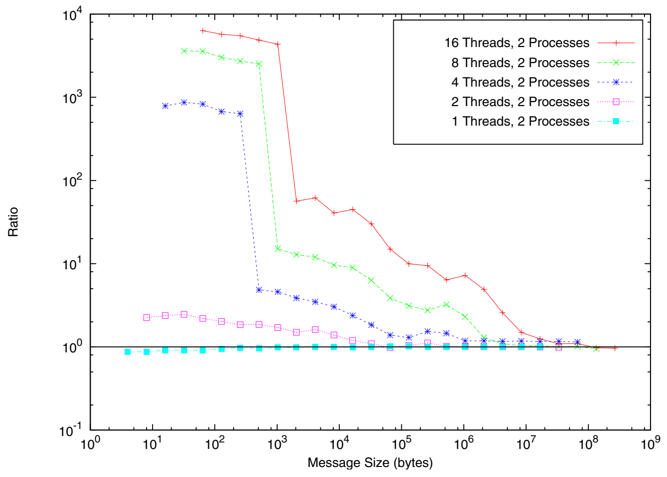


Fig. 8. IBM Power5: Ratio of Multiple to Master-only PingPong execution times

one OpenMP thread per MPI process. The MPI ranks participating in the benchmark are chosen to lie on different nodes. On both systems, for small data sizes, the execution time is least for one thread per MPI process, and increases with the number of threads, whereas for large data sizes, the execution time is greatest for one thread per MPI process, and decreases with the number of threads. The crossover between these regimes occurs between 10^3 and 10^4 bytes. Recall that the send and receive buffers are being written/read by multiple threads. For small data sizes, the overhead of parallelisation is not

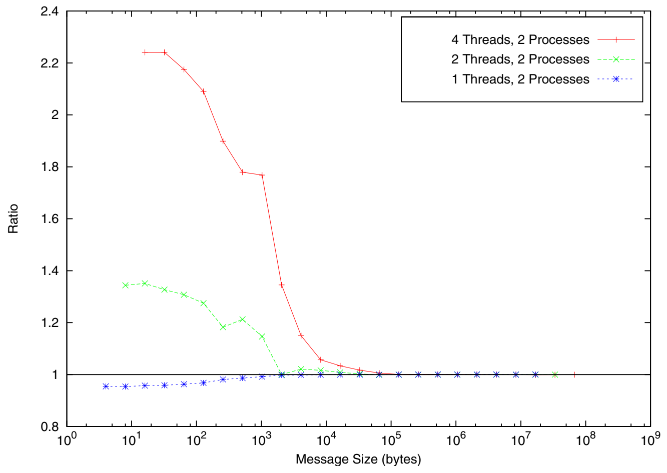


Fig. 9. IBM BlueGene/P: Ratio of Multiple to Master-only PingPong execution times

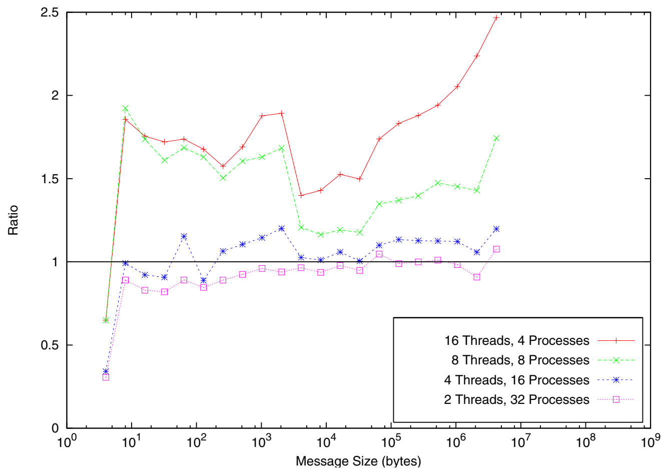


Fig. 10. IBM Power 5: Ratio of execution time for Reduce benchmark to one thread per process

worthwhile, but above the crossover, significant benefit is gained from having multiple threads employed. The other hardware platforms display similar behaviour (not shown here).

Figures 8 and 9 show the results of running the Multiple version of the PingPong benchmark on the IBM Power 5 cluster and BlueGene/P system respectively. In this case the execution times are normalised by the time for the Master-only PingPong benchmark running on the same number of processes and threads. For the Power5 system, we observe very poor performance for the

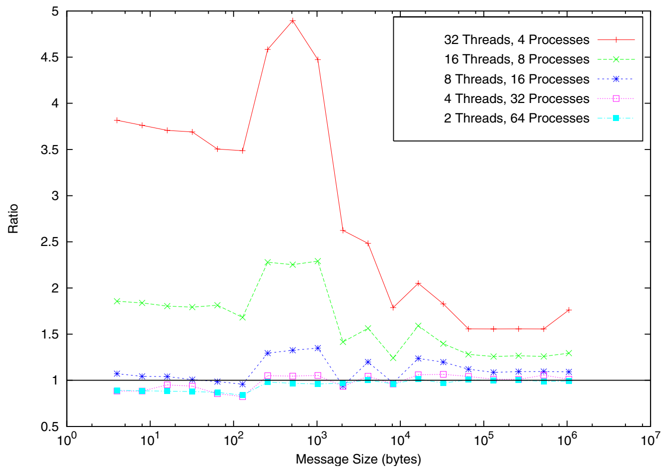


Fig. 11. IBM Power 6: Ratio of execution time for Reduce benchmark to one thread per process

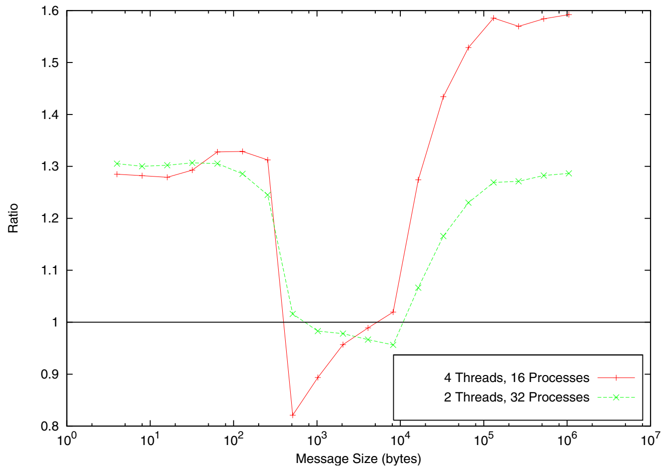


Fig. 12. IBM BlueGene/P: Ratio of execution time for Reduce benchmark to one thread per process

Multiple version on small data sizes: in some cases it is over 3 orders of magnitude slower (note the log scale on the vertical axis in Figure 8. The Power 6 system displays similar behaviour to the Power 5: contention for locks inside the MPI library is a possible cause of this. In contrast, the Multiple version on the BlueGene/P system is a little over two times slower using four threads per process than using one. On neither system is there any benefit gained from calling MPI from multiple threads for large data sizes. This suggests that a single large message is able to utilise all the off-node bandwidth.

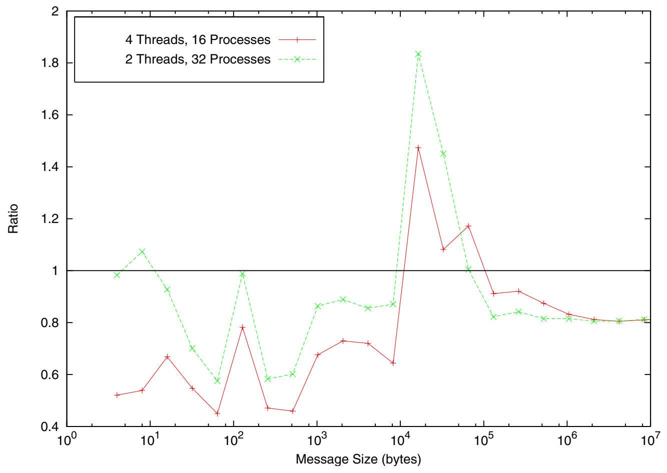


Fig. 13. Cray XT4: Ratio of execution time for Reduce benchmark to one thread per process

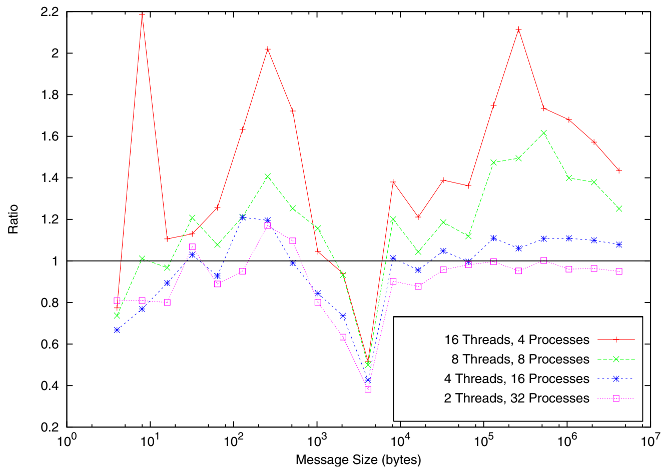


Fig. 14. IBM Power 5: Ratio of execution time for AlltoAll benchmark to one thread per process

Figures 10 to 13 show the results of running the Reduce benchmark on all four platforms. The execution times are normalised to the execution time with one OpenMP thread per MPI process.

On the IBM Power 5 and Power 6 systems, we observe that the mixed-mode version of Reduce is generally slower than the pure MPI (one thread per process), though there are some modest gains to be had by using two threads per process for small data sizes. On the BlueGene/P system, the mixed-mode version is

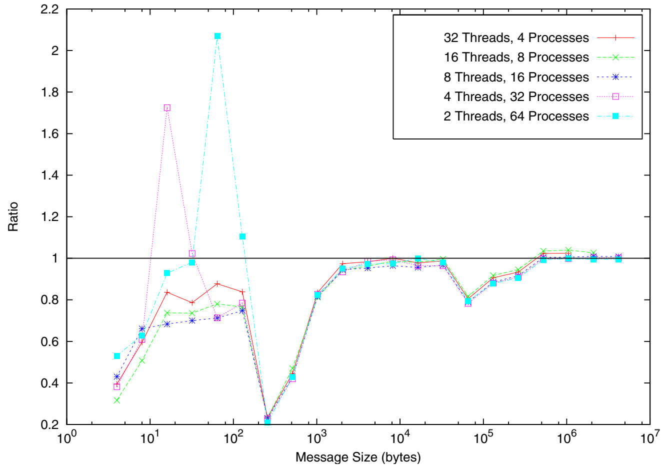


Fig. 15. IBM Power 6: Ratio of execution time for AlltoAll benchmark to one thread per process

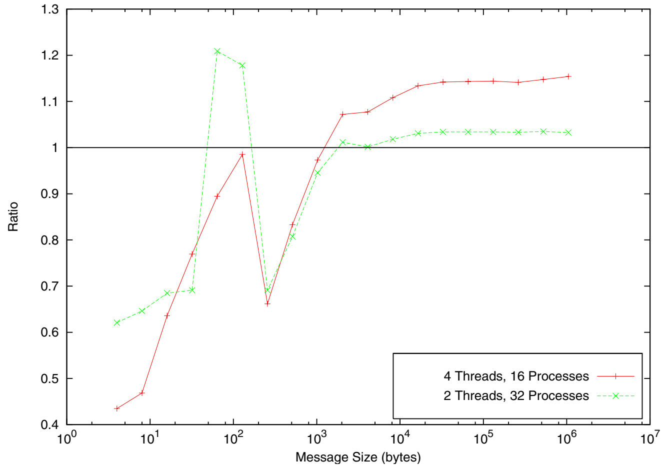


Fig. 16. IBM BlueGene/P: Ratio of execution time for AlltoAll benchmark to one thread per process

also slower, except for a window of data sizes between 10^3 and 10^4 bytes. This suggests that on these platforms, the MPI Reduce is well optimised for shared memory nodes. It is also possible that the implementation of OpenMP array reductions is not very efficiently implemented. On the Cray XT4, however, the mixed mode version is generally faster, except for data sizes between 10^4 and 10^5 bytes. This system is known to suffer from contention between cores on the same node for access to the network: having fewer, larger, messages entering the network seems to be beneficial.

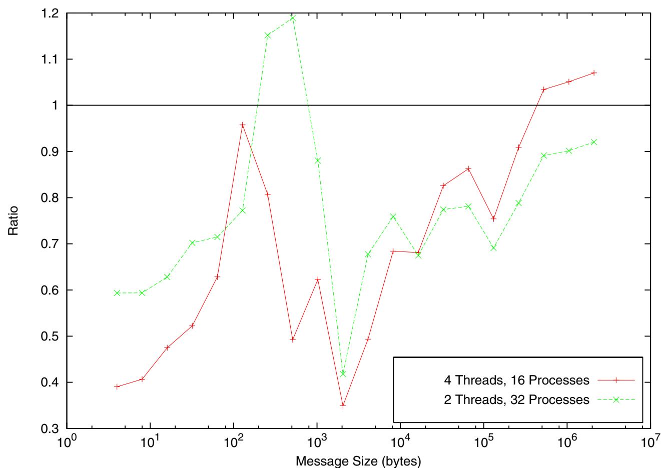


Fig. 17. Cray XT4: Ratio of execution time for AlltoAll benchmark to one thread per process

Figures 14 to 17 show the results of running the AlltoAll benchmark on all four platforms. The execution times are normalised to the execution time with one OpenMP thread per MPI process. The four systems show different behaviours for this benchmark. On the IBM Power 5, the mixed mode version is significantly faster for data sizes in the range 10^3 to 10^4 bytes, and the optimal number of threads per process is usually two. For the IBM Power 6, having multiple threads per process is beneficial on small data sizes, but increasing the number of threads per process beyond two makes little difference. On the BlueGene/P, mixed-mode is worthwhile for small data sizes, but not large ones, and on the Cray XT4 it is worthwhile for almost all data sizes and is up to three times faster in some cases.

4 Conclusions and Future Work

We have described the design and implementation of a set of microbenchmarks for mixed-mode OpenMP/MPI programming. These cover both point-to-pont and collective communication patterns. We have run these benchmarks on four current HPC architectures: the results show some interesting performance differences between the architectures and highlight some possible inefficiencies in the implementation of MPI on these systems.

In the future, we intend to run the benchmarks on other systems, for example on Intel- and Opteron-based clusters (where there may be multiple combinations of MPI library and OpenMP compiler available) and on vector systems such the NEC SX-9 and the Cray X2. We can also consider additions to the benchmark suite: for example multi-PingPong (where every core on a node communicates with a corresponding core on another node).

References

1. Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. In: Proceedings of the Third European Workshop on OpenMP (EWOMP 2001), Barcelona, Spain (September 2001)
2. Hutter, J., Curioni, A.: Dual-level Parallelism for Ab Initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code. Parallel Computing 31(1), 1–17 (2005)
3. Intel. MPI Benchmarks, `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm`
4. MPI Forum, MPI: A Message-Passing Interface Standard Version 2.1 (2008)
5. OpenMP ARB, OpenMP Application Programming Interface Version 3.0 (2008)
6. Rabenseifner, R.: Hybrid Parallel Programming on HPC Platforms. In: Proceedings of the Fifth European Workshop on OpenMP, EWOMP 2003, Aachen, Germany, September 22-26, pp. 185–194 (2003)
7. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2009 (2009) (to appear)
8. Reussner, R., Sanders, P., Traeff, J.L.: SKaMPi: a Comprehensive Benchmark for Public Benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002)
9. Salmond, D., Saarinen, S.: Early Experiences with the New IBM p690+ at ECMWF. In: Proceedings of the Eleventh ECMWF Workshop Reading, UK, pp. 1–12. World Scientific, Singapore (2005)
10. Smith, L., Bull, M.: Development of Mixed Mode MPI/OpenMP Applications. Scientific Programming 9(2-3), 83–98 (2001)
11. The Sphinx Parallel Microbenchmark Suite, `http://www.llnl.gov/CASC/sphinx/sphinx.html`

Performance Profiling for OpenMP Tasks

Karl Fürlinger[1] and David Skinner[2]

[1] Computer Science Division, EECS Department
University of California at Berkeley
Soda Hall 593, Berkeley CA 94720, U.S.A.
`fuerling@eecs.berkeley.edu`
[2] Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley CA 94720, U.S.A.
`deskinner@lbl.gov`

Abstract. Tasking in OpenMP 3.0 allows irregular parallelism to be expressed much more easily and it is expected to be a major step towards the widespread adoption of OpenMP for multicore programming. We discuss the issues encountered in providing monitoring support for tasking in an existing OpenMP profiling tool with respect to instrumentation, measurement, and result presentation.

1 Introduction

The direct support for task parallelism in version 3.0 of the OpenMP standard is expected to be a major step towards the widespread adoption of OpenMP for shared memory multicore programming. Tasking allows irregular forms of parallelism to be expressed more easily and it will allow OpenMP to be employed in new application areas.

In this paper we discuss the issues we encountered in providing monitoring support for tasking in the `ompP` profiling tool with respect to instrumentation and measurement and result presentation. Since tasking results in more dynamic and unpredictable execution characteristics of OpenMP codes, we believe tool support will be more important for users that would like to understand how their code executes and what performance it achieves. As an example, the OpenMP v3.0 specification states that, when a thread encounters a task construct, "[it] may immediately execute the task, or defer its execution". To some application developers it will be important to know what decision the runtime took and `ompP`'s profiles offer this kind of information, among other things.

The rest of this paper is organized as follows: in Sect. 2 we give a short overview of the OpenMP profiling tool we have extended in this study to support tasking. In Sect. 3 we describe the extensions and modifications made, at the instrumentation, measurement, and result presentation stages. In Sect. 4 we discuss related work and in Sect. 5 we conclude and discuss areas for future work.

2 The OpenMP Profiler ompP

ompP is a profiling tool for OpenMP applications that does not rely on nor is limited to a particular OpenMP compiler and runtime system. ompP differs from other profiling tools like gprof or OProfile [6] in primarily two ways. First, ompP is a measurement-based profiler and does not use program counter sampling. The application with source code instrumentation invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). The direct measurement approach can potentially lead to higher overheads when events are generated very frequently, but this can be avoided by instrumenting such constructs selectively. An advantage of the direct approach is that the results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts.

The second difference lies in the way of data collection and representation. While general profilers work on the level of routines, ompP collects and displays performance data in the user model of the execution of OpenMP events [5]. For example, the data reported for critical sections contain not only the execution time but also list the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each threads spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile for a critical section is given in Fig. 1.

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT
   0       1.00          1       1.00       0.00       0.00
   1       3.01          1       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       0.00
   3       4.01          1       1.00       3.01       0.00
 SUM      10.02          4       4.01       6.01       0.00
```

Fig. 1. Profiling data delivered by ompP for a critical section

Profiling data in a similar style is also delivered for other OpenMP constructs, the columns (execution times and counts) depend on the particular construct. Furthermore, ompP supports the query of hardware performance counters through PAPI [3] and the measured counter values appear as additional columns in the profiles.

Profiling data are displayed by ompP both as flat profiles and as callgraph profiles, giving both inclusive and exclusive times in the latter case. The callgraph profiles are based on the callgraph that is recorded by ompP. An example callgraph is shown in Fig. 2. The callgraph is largely similar to the callgraphs given by other tools, such as callgrind [9], with the exception that the nodes are not only functions but also OpenMP constructs and user-defined regions, and the (runtime) nesting of those constructs is shown in the callgraph view. The callgraph that ompP records represents the union of the callgraph of each thread. That is, each node reported has been executed by at least one thread.

```
    ROOT  [critical.i686.ompp: 4 threads]
  REGION  +-R00004 main.c (40-51) ('main')
PARALLEL     +-R00005 main.c (44-48)
  REGION        |-R00001 main.c (20-22) ('foo')
  REGION        |  +-R00002 main.c (27-32) ('bar')
CRITICAL        |     +-R00003 main.c (28-31) (unnamed)
  REGION        +-R00002 main.c (27-32) ('bar')
CRITICAL           +-R00003 main.c (28-31) (unnamed)
```

Fig. 2. Example callgraph view of `ompP`

3 Supporting Tasks in `ompP`

The OpenMP 3.0 specification introduces two new constructs for tasking, `task` and `taskwait`. If a thread encounters a `task` construct, it packages up the code and data environment and creates the task to be executed in the future, potentially by a different thread. The `taskwait` construct is used to synchronize the execution of tasks. A thread can suspend the execution only at a task scheduling point (TSP). The same thread will pick up the execution of a task, unless the task is untied. In this case, any thread can resume the execution and no restriction on the location of TSPs in untied tasks exists.

3.1 Instrumentation

`ompP` relies on source code instrumentation using Opari [7] to add monitoring calls according to the POMP specification inside and around OpenMP constructs. We extended Opari to handle the `task` and `taskwait` constructs as described below.

For `task`, an instrumented piece of code looks similar to the pseudocode depicted in Fig. 3. I.e., `enter/exit` instrumentation calls are placed on the outside of the task construct and `begin/end` calls are placed as the first and last statements inside the tasking code, respectively.

If specified, the untied clause is detected and `POMP_Utask_*` calls are generated in this case. A simple enter/exit pair of instrumentation calls is added for the taskwait clause.

3.2 Measurement

`ompP`'s measurement routines implement the `POMP_Task_*`, `POMP_Utask_*`, and `POMP_Taskwait_*` calls. An important observation is that during execution a task construct is best represented by two separate entities: one for task creation and one for task execution. Following this idea, we create two `ompP` regions for each source code task construct, one of type `TASK` for task creation and one of type `TASKEXEC` to record profiling data related to task execution. In the terminology of the OpenMP specification, `TASK` corresponds to the task construct, while `TASKEXEC` corresponds to the task region. `UTASK` and `UTASKEXEC` are used for untied tasks.

```
POMP_Task_enter(...)
#pragma omp task
{
POMP_Task_begin(...)
// user's task code
POMP_Task_end(...)
}
POMP_Task_exit(...)
```

(a) POMP instrumentation for the `task` construct.

```
POMP_Taskwait_enter(...)
#pragma omp taskwait
POMP_Taskwait_exit(...)
```

(b) POMP instrumentation for the `taskwait` construct.

Fig. 3. Instrumentation for tasking related constructs

```
void main(int argc, char* argv[]) {
#pragma omp parallel {
    int i;
#pragma omp single nowait {
      for( i=0; i<5; i++ ) {
#pragma omp task /* if(0) */ {
       mytask();
      }
    }
  }
}
```

```
void mytask() {
  sleep(1);
 }
```

Fig. 4. (Pseudo) source code with tasking

Consider the simple code example in Fig. 4 and its corresponding callgraph as delivered by `ompP` in Fig. 5. Task creation occurs inside the single region while task get executed when threads hit the implicit barrier at the end of the parallel construct. If, alternatively, an `if(0)` clause is specified, tasks are executed immediately and this is visible in the callgraph, where the `TASKEXEC` is a child of the `TASK` node.[1]

Support for monitoring untied tasks is incomplete at this time. We chose to offer the user the option to disable any monitoring of untied tasks completely or to monitor them in the same way as tied tasks (assuming that the executing thread does not change during the lifetime of a task). Without a way to observe the suspension and resumption of tasks at general task scheduling points, this seems to be the best we can do.

[1] All experiments reported in this paper have been performed on a Linux machine using a beta version of Intel's C/C++ compiler suite v11.0.044, which supports tasking. We suspect this implementation might be not be fully optimized but it was a sufficient as a vehicle to test the feasibility of our monitoring approach, as this paper is concerned about functionality and not performance.

```
PARALLEL  +-R00001          |  PARALLEL  +-R00001 main.c (15-26)
  SINGLE     |-R00002       |    SINGLE     +-R00002 main.c (18-25)
    TASK     |  +-R00003    |      TASK        +-R00003 main.c (22-23)
TASKEXEC     +-R00003       |  TASKEXEC           +-R00003 main.c (22-23)
```

Fig. 5. Dynamic callgraph of the code shown in Fig. 4 (left). The right side shows the callgraph when an `if()` clause is present and evaluates to false.

3.3 Profiling Data Analysis and Presentation

Flat profiles and callgraph profiles are recorded for the `[U]TASK`, `[U]TASKEXEC`, and `TASKWAIT` regions. Fig. 6 shows the possible immediate dynamic nesting of these three region types and their interpretation. The location of the `[U]TASKEXEC` region in the callgraph allows the analysis of when tasks were executed dynamically. The execution might happen nested in the `[U]TASK` region if an `if()` clause evaluates to false. If tasks are executed at a `TASKWAIT` region, this will also be indicated by the dynamic nesting and if threads execute tasks while at the implicit exit barrier of a parallel or workshare construct, the `TASKEXEC` region will be shown as a child region of this parallel or workshare region.

	inner region		
outer region	`[U]TASK`	`TASKEXEC`	`TASKWAIT`
`[U]TASK`	–	× (see caption)	–
`TASKEXEC`	×	× (task-switching)	×
`TASKWAIT`	–	×	–

```
+-....
|- outer
   |- inner
   +-....
```

Fig. 6. Possible nesting of the tasking related region types in `ompP`. A dash symbol (–) indicates a nesting that can not occur, × indicates valid nestings. The `[U]TASK`–`TASKEXEC` nesting signifies immediate execution either because the `if()` clause evaluates to false or runtime decided not to defer the execution for other reasons such as resource exhaustion.

One of `ompP`'s more advanced features is its overhead analysis. When threads execute a worksharing region with an imbalanced amount of work, the waiting time of threads in the implicit exit barrier of that worksharing construct is measured by `ompP` and reported as load imbalance overhead. A total of four overhead classes are defined: load imbalance; synchronization overhead; limited parallelism; and thread management. The reporting of the overhead relies on the fact that OpenMP threads do not perform useful work on behalf of the application in certain program phases (such as when entering a critical section or at implicit or explicit thread barriers).

This assumption is no longer valid with OpenMP 3.0 when tasking is used. When threads hit an implicit barrier, instead of idling they can do useful work by executing ready tasks. To account for this, we modified the overhead reporting of `ompP` by subtracting from the overheads the time spent executing tasks. The required timing data is available from the callgraph recorded by `ompP` (we know

```
Overheads wrt. each individual parallel region:
      Total   Ovhds (%) =  Synch(%) + Imbal  (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (16.68) 0.00 (0.00) 1.00 (16.66)  0.00 (0.00) 0.00 (0.02)

Overheads wrt. whole program:
      Total   Ovhds (%) =  Synch(%) + Imbal  (%) + Limpar (%) +  Mgmt (%)
R00001 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
   SUM 6.00 1.00 (15.64) 0.00 (0.00) 1.00 (15.63)  0.00 (0.00) 0.00 (0.02)
```

Fig. 7. Overhead analysis report corresponding to the code shown in Fig. 4

```
R00001 main.c (15-26) PARALLEL
 TID    execT    execC     bodyT  exitBarT  startupT  shutdwnT     taskT
   0     3.00        1      0.00      0.00      0.00      0.00      3.00
   1     3.00        1      0.00      1.00      0.00      0.00      2.00
 SUM     6.00        2      0.00      1.00      0.00      0.00      5.00
```

Fig. 8. Flat region profile, showing the time threads spend executing tasks while waiting at the implicit exit barrier of the parallel region. This data corresponds to the code shown in Fig. 4 when executed with two threads.

that a `[U]TASKEXEC` happens in the context of the implicit exit barrier) and from the callgraph profiles recorded for the task execution on a per-thread basis.

An example of an overhead report that takes task execution into account is shown in Fig. 7. This overhead report corresponds to the code fragment shown in Fig. 4, the application executes with two threads and creates 5 tasks with an execution time of 1 second each. As shown, ompP correctly accounts for the task execution by reporting the imbalance overhead as 1.0 second due to the uneven distribution of tasks to threads.

To allow the application developers to analyze when tasks get executed further, we added a new timing category `taskT` to OpenMP parallel regions and worksharing regions. Fig. 8 shows the profile of a parallel region. While at the implicit exit barrier of the parallel construct, thread 0 spent 3.0 seconds executing tasks, while thread 1 spent 2.0 seconds, 1.0 second remains as the waiting time of thread 1, as shown in the `exitBarT` column.

4 Related Work

Opari and the POMP interface are the basis of OpenMP monitoring for several performance tools for scientific computing like TAU [8], KOJAK [10], and Scalasca [4]. To the best of our knowledge, there is currently no work under way to support tasking within these projects [2]. However, we believe that our work on extending Opari and the experience we gathered with respect to supporting tasking in the monitoring system will be of use for adding tasking support for these tools.

Sun has developed an extension to their proposed performance profiling API [5] for OpenMP and is supporting tasking in the new version of their performance tool suite. [1]. The nature of this interface and Sun's implementation are different from ompP's approach (callbacks and sampling vs. direct measurement).

5 Conclusion and Future Work

We have described our experiences in supporting tasking in a measurement based profiler for OpenMP. We have made additions to a source code instrumentation, measurement, and result presentation stages of the tool.

With respect to measurement, a fundamental difference arose in the way waiting time at implicit barriers was accounted for in the overhead analysis. The modified data reporting allows users to see which threads execute tasks at which point in the application. Due to the dynamic execution characteristics of OpenMP with tasking, we believe this capability is important both for performance considerations as well as a pedagogical tool for people learning to use OpenMP tasking.

We found that monitoring overhead directly correlates with the frequency of monitored events. With very frequent, short lived tasks overheads can be substantial. However, such an application is unlikely to scale or to perform well even without any monitoring. For reasonably sized tasks we found that monitoring overhead can be expected to be less than 5 percent of execution time.

For the future, work is planned in several directions. Clearly, how untied tasks are handled currently is unsatisfactory. However, without notifications of task switches form the runtime, the options for a source code instrumentation based tool like ompP are very limited. The most promising solution for this issue seems to lie in an incorporation of the profiling API [5] for providing such a notification mechanism. The currently limited adoption of this API by vendors is a practical problem, however.

References

1. OpenMP 3.0: Ushering in a new era of parallelism. Birds of a Feather meeting at Supercomputing (2008)
2. Personal communication at supercomputing (2008)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.J.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. 14(3), 189–204 (2000)
4. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: Scalable parallel trace-based performance analysis. In: Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006), Bonn, Germany, pp. 303–312 (2006)
5. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling. Accepted by the OpenMP ARB as an official ARB White Paper, `http://www.compunity.org/futures/omp-api.html`

6. Levon, J.: OProfile, A system-wide profiler for Linux systems, `http://oprofile.sourceforge.net`
7. Mohr, B., Malony, A.D., Shende, S.S., Wolf, F.: Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In: Proceedings of the Third Workshop on OpenMP (EWOMP 2001) (September 2001)
8. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications, ACTS Collection Special Issue (2005)
9. Weidendorfer, J., Kowarschik, M., Trinitis, C.: A tool suite for simulation based analysis of memory access behavior. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 440–447. Springer, Heidelberg (2004)
10. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. In: Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003), February 2003, pp. 13–22. IEEE Computer Society, Los Alamitos (2003)

Tile Reduction: The First Step towards Tile Aware Parallelization in OpenMP

Ge Gan[1], Xu Wang[2], Joseph Manzano[1], and Guang R. Gao[1]

[1] Dep. of Electrical and Computer Engineering
University of Delaware
Newark, Delaware 19716 U.S.A.
{gan,jmanzano,ggao}@capsl.udel.edu
[2] Dep. of Electrical Engineering
Jilin University
Jilin, China 130000
wangxu@ict.ac.cn

Abstract. Tiling is widely used by compilers and programmer to optimize scientific and engineering code for better performance. Many parallel programming languages support tile/tiling directly through first-class language constructs or library routines. However, the current OpenMP programming language is *tile oblivious*, although it is the *de facto* standard for writing parallel programs on shared memory systems. In this paper, we introduce *tile aware parallelization* into OpenMP. We propose *tile reduction*, an OpenMP tile aware parallelization technique that allows reduction to be performed on multi-dimensional arrays. The paper has three contributions: **(a)** it is the first paper that proposes and discusses tile aware parallelization in OpenMP. We argue that, it is not only necessary but also possible to have tile aware parallelization in OpenMP; **(b)** the paper introduces the methods used to implement tile reduction, including the required OpenMP API extension and the associated code generation techniques; **(c)** we have applied tile reduction on a set of benchmarks. The experimental results show that tile reduction can make parallelization more natural and flexible. It not only can expose more parallelism in a program, but also can improve its data locality.

1 Introduction

Tiling [1] [2] has been used as an effective compiler optimizing technique to generate high performance scientific codes. Tiling not only can improve data locality for both the sequential and parallel programs [3] , but also can help the compiler to maximize parallelism and minimize synchronization [4] for programs running on parallel machines. Thus, sometimes, it is used by the programmers to hand-tune their scientific programs to get better performance.

Tiling is essentially a program design paradigm. It is a natural representation for many important data objects that are heavily used in scientific and engineering algorithms. Scientific code that is written with the concept of tile/tiling in mind usually looks concise and clear, and thus is much easier to understand and less error prone.

Due to these advantages, it is desirable to provide certain high level language constructs in the programming languages to support tile/tiling in program design directly. To meet this requirement, researchers have proposed various designs in many parallel programming languages or sublanguages. The examples include HPF[5], UPC[6], X10[7], ZPL[8], CAF[9], Titanium[10], and HTA[11], which are among the most popular parallel languages. However, it is interesting to find out that, in the current OpenMP APIs, no directive or clause can be used to annotate data tiles and carry such information to the OpenMP compiler. In other words, the current OpenMP programming language is *tile oblivious*, although it is the *de facto* standard for writing parallel programs on shared memory systems.

In this paper, we propose *tile aware parallelization* for the OpenMP programming language. Its purpose is to enhance the OpenMP API with the concept of tile/tiling so that more data parallelism can be exposed to the OpenMP compiler. Besides granting greater flexibility to the OpenMP compiler to perform more data parallelization, it brings better data locality into the code. This is achieved by extending the current OpenMP directives, clauses, and runtime routines, or introducing new language constructs into OpenMP. Our first effort in this direction is termed *tile reduction*, an OpenMP tile aware parallelization technique that allows parallel reduction to be performed on multi-dimensional arrays.

Reduction is a form of recursive calculation that use mathematically associative and commutative operators to "aggregate" a set of data. Reduction can be performed in parallel to improve performance. For this reason, many programming languages and sub-languages support parallel reduction. Some examples are UPC [12], MPI [13], ZPL [14], and OpenMP [15]. According to the current OpenMP API specification, reduction can only be performed on "named scalar" variables. It cannot be applied on multi-dimensional arrays. We call this kind of reduction *scalar reduction*. In this paper, we introduce a new technique called *tile reduction*, which evolves the current reduction parallelization from scalar variables to multi-dimensional arrays. We have extended the traditional `reduction` clause to allow the programmers to annotate their code where tile reduction can be applied. We have also developed the required code generation technique to interpret the new `reduction` clause and generate the required parallel code accordingly. The major contributions of this paper are:

1. As far as the authors are aware, this is the first paper that proposes and discusses tile aware parallelization in OpenMP. We argue that, it is not only necessary but also possible to have tile aware parallelization techniques in OpenMP.
2. The paper introduces tile reduction, an OpenMP tile aware parallelization technique that applies reduction on multi-dimensional arrays. We discuss the methods used to implement tile reduction, including the required OpenMP API extension and the associated code generation technique.
3. We evaluate the tile reduction technique with a set of benchmarks. The experimental results show that using tile reduction can make the code parallelization more natural and flexible. It not only can expose more parallelism in the program but also can improve its data locality.

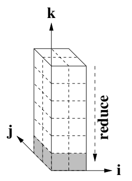
The rest of the paper is organized as follows. In Section 2, we use a motivating example to show why tile reduction is necessary. Section 3 will discuss how to implement tile reduction in the OpenMP compiler. We present our experimental data in Section 4 and make our conclusions in Section 5.

2 Motivation

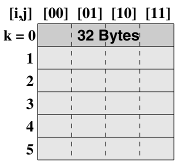
In this section, we use the ”histogram reduction” [16] code as an example to demonstrate the limits of the current OpenMP reduction clause. We will also use the same example to show the advantages of extending *scalar reduction* to *tile reduction*.

```
long long A[][2][2];
    ...
for (k=1; k<10000000; k++)
  for (i=0; i<2; i++)
    for (j=0; j<2; j++)
      A[0][i][j] += A[k][i][j]
```

(a) Original Histogram Reduction Code



(b) The 3D Diagram



(c) A’s Memory Layout

Fig. 1. The Histogram Reduction Example

Figure 1(a) shows the code of the histogram reduction program. The code works on `A[][][]`, a 3-dimensional array with each element containing an 8-byte `long long`. It aggregates all elements along the `k` dimension and stores the results in the `2x2` tile `A[0][][]`. The diagram in Figure 1(b) shows these operations. We assume that the cache line size is 32 bytes and that the the array is stored in a row-major order in the memory. Therefore, elements with the same `k` coordinate can be fed into the same cache line, as shown in Figure 1(c). There are three nested loops in the code. Each loop traverses one of the `i`, `j`, `k` dimension of the array. Data dependence only exit in loop `k` because of the recursive calculation.

```
for (k=1; k<10000000; k++)
  #pragma omp parallel for
  for (i=0; i<2; i++)
    for (j=0; j<2; j++)
      A[0][i][j] += A[k][i][j]
```

(a) Parallelize loop ”i”

```
for (k=1; k<10000000; k++)
  #pragma omp parallel for collapse(2)
  for (i=0; i<2; i++)
    for (j=0; j<2; j++)
      A[0][i][j] += A[k][i][j]
```

(b) Parallelize loop ”i” and ”j” using the collapse clause

Fig. 2. Parallelize the Histogram Reduction Program Without Changing the Code

Given the code in Figure 1(a), there are many different ways to parallelize it. However, due to the data dependence in loop `k`, we cannot parallelize this loop. Therefore, without changing the code, we can only parallelize loop `i` and `j`, as shown in Figure 2(a) and 2(b). It is obvious that there are trivial workload and little parallelism in loop `i` and loop `j`. Thus, it is not worthwhile to parallelize these two loops, even while using the `collapse` clause (supported in OpenMP 3.0 [15]).

To get a larger workload and more parallelism, we can interchange the loops manually before parallelizing the code, as shown in Figure 3. In Figure 3(a) and 3(b), the

```
#pragma omp parallel for
for (j=0; j<2; j++)
  for (i=0; i<2; i++)
    for (k=1; k<10000000; k++)
      A[0][j][i] += A[k][j][i]
```

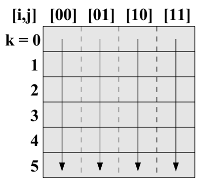
(a) Parallelize the outer loop

```
#pragma omp parallel for collapse(2)
for (j=0; j<2; j++)
  for (i=0; i<2; i++)
    for (k=1; k<10000000; k++)
      A[0][j][i] += A[k][j][i]
```

(b) Parallelize the outer two loops

```
#pragma omp parallel for private(sum) collapse(2)
for (j=0; j<2; j++)
  for (i=0; i<2; i++) {
    sum = 0; #pragma omp parallel for shared(sum)
reduction(+:sum)     for (k=0; k<10000000; k++)
      sum += A[k][j][i]
    A[0][j][i] = sum;
  }
```

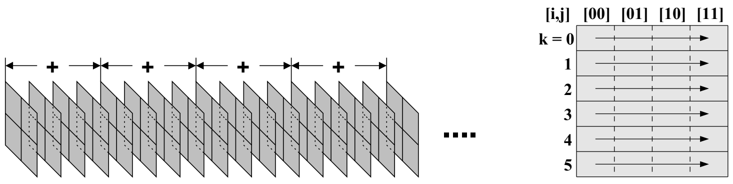
(c) Nested parallelization to harvest more parallelism



(d) Data access pattern

Fig. 3. More Parallelization for Histogram Reduction Code

workload that can be assigned to the threads is large enough. However, the available parallelism is still very small (only supports two or four concurrent threads). Figure 3(c) shows a better solution. In Figure 3(c), a nested `parallel for` directive is used to parallelize the recursive addition using the `reduction` clause (with trivial code change). Although the code in Figure 3(c) can leverage all levels of parallelism in the program, its strided data access pattern would cause a great number of unnecessary cache misses, as shown in Figure 3(d). Code in Figure 3(a) and 3(b) have the same data locality problem. Apparently, the current OpenMP parallelization techniques cannot harvest the maximum parallelism and data locality in the code at the same time. They suffer from either insufficient parallelism or poor data locality.



(a) Schema of tile reduction

(b) Better locality

Fig. 4. The Ideal Parallelization Schema for the Histogram Reduction Code

The ideal parallelization is shown in Figure 4. Logically, the recursive addition can be viewed as being performed on an array of `2x2` data tiles. In theory, these tiles can be added together in parallel by multiple threads, as shown in Figure 4(a). In this way, the code can achieve both the maximum parallelism and the best data locality (see Figure 4(b)). Besides, from the programmers' angle, this is the most natural way to perform parallelization on this piece of code. However, the current OpenMP specification does not provide any mechanism to support such kind of parallelization. This motivates us to extend the traditional *scalar* reduction to *tile* reduction.

3 Tile Reduction

In this section, we will discuss the techniques used to implement tile reduction. They include the extended OpenMP programming interface and the required code generation design. The related runtime support will be mentioned when needed.

3.1 Programming Interface Extension

In order to support tile reduction, we need to extend the current OpenMP programming interface. The extension was made based on three criteria. First, it must be able to cover most of the common cases of tile reduction code. Second, it must be simple and easy to use and provide the programmers with the maximal flexibility. Third, the extension should not complicate the code generation of the OpenMP compiler and the OpenMP runtime. Figure 5(a) shows the OpenMP API (C/C++) extension we proposed for the `reduction` clause. Figure 5(b) gives a simple example that uses the extended `reduction` clause to parallelize the tile reduction code.

`reduction(operator : T`$[j_k, L_k, U_k] ... [j_2, L_2, U_2][j_1, L_1, U_1]$`)`

`T`:	Tile name
k:	Dimension of the tile
j_i:	the loop index that is used in the traversal of the i^{th} dimension of the tile
L_i:	the lower bound of j_i
U_i:	the strict upper bound of j_i

(a) OpenMP API (C/C++) extension for the `reduction` clause

```
int B[2][2] = {{0,0},{0,0}};
        ...
#pragma omp parallel for reduction(+: B[j,0,2][i,0,2])
for (k=0; k<10000000; k++)
  for (j=0; j<2; j++)
    for (i=0; i<2; i++)
      B[j][i] += A[k][j][i]
```

(b) Simple example using the extended API

Fig. 5. OpenMP API (C/C++) extension and a simple example code

Compared with the current OpenMP API specification, the difference is in the `list` construct. In addition to the ”named scalar” variables, we allow the programmers to put a ”multi-dimensional array” in the `list` construct. This ”multi-dimensional array” is not a real array data structure in the language sense. It is a language construct that conveys important information to the OpenMP compiler. It tells the compiler the shape, the size, and the element type of the tile and how its elements are traversed by the loops.

To make the paper easy to follow, we call the tile under reduction as the *reduction tile*; the ”multi-dimensional array” in the `list` construct as the *tile descriptor*; and the loops involved in performing ”one” recursive calculation as the *reduction kernel loops*. For the example in Figure 5(b)[1], the reduction tile is `B[][]`, the tile descriptor

[1] Index variable k starts from zero because array B[][] is used to store the accumulation results, otherwise it starts from one.

is `B[j,0,2][i,0,2]`, and the reduction kernel loops are the `j` and `i` loops (not including the `k` loop, i.e., the parallelized loop). In our design, the shape of the reduction tile must be a rectangle or a high-dimensional rectangle. Triangle or other shapes are not yet supported. The exact shape and size of the reduction tile are determined by the tile descriptor.

The format of the tile descriptor is shown in Figure 5(a). It has two parts: the *tile name* (i.e., `T`) and the *dimension descriptor* (i.e., $[j_k, L_k, U_k]...[j_2, L_2, U_2][j_1, L_1, U_1]$). Tile name must be the same as the multi-dimensional array variable on which the recursive calculations are performed. For the example in Figure 5(b), this corresponds to the name of the *lhs* variable in line 4, which is `B`. It tells the OpenMP compiler the data type of the tile element, which must be a built-in scalar type. The dimension descriptor, on the other hand, is an array of 3-tuples. Each 3-tuple corresponds to one dimension of the tile and stores important information of that dimension. These 3-tuples are listed in the dimension descriptor in descendant order (higher dimension first). Each 3-tuple has three elements: loop index variable, upper bound expression, and lower bound expression. The loop index variable identifies a loop in the reduction kernel loops. Since stride accesses are not allowed, the loop stride is always 1, so it is omitted from the tuple. The size of the k-dimensional tile is calculated from equation (1).

$$(U_k - L_k) \times ...(U_2 - L_2) \times (U_1 - L_1) \tag{1}$$

The information stored in the tile descriptor is very important for the OpenMP compiler to generate correct parallel code.

The `operator`, as usual, must be a mathematically associative and commutative operator that performs the recursive calculation. In our current example, it is a "+".

```
#pragma omp parallel for reduction(+: A[j,0,2][i,0,2])
for (k=1; k<10000000; k++)
 for (j=0; j<2; j++)
  for (i=0; i<2; i++)
    A[0][j][i] += A[k][j][i]
```

Fig. 6. Tile reduction: tile is part of a bigger multi-dimensional array

The reduction tile is not required to be a standalone multi-dimensional array. Instead, it can be part of another larger multi-dimensional array. For example, in the code in Figure 6, the reduction tile is `A[0][j][i]` ($j = \{0, 1\}, i = \{0, 1\}$). It is a 2×2 slice cut out from the 3-dimensional array `A[][][]`;

Besides, as we have mentioned before, the lower and upper bounds in the dimension descriptor are expressions. They are not required to be constants. Generally, the lower and upper bounds can be a function of other variables, as long as the result of the function can be decided at runtime. Figure 7 shows such an example. The code in Figure 7 is a blocked matrix multiplication program. It is easy to see that there is an opportunity to apply tile reduction on the loop in line 3, i.e., the `kk` loop. The diagram on the right hand side gives an intuitive illustration. In this example, the reduction tiles are blocks cut out from a big 2×2 matrix (`C[][]`). Therefore, the lower and upper bounds of the

```
0 for (ii=0; ii<n; ii+=b)
1  for (jj=0; jj<n; jj+=b)
2 #pragma parallel for reduction(+: \
  C[i,ii,min(ii+b,n)][j,jj,min(jj+b,n)])
3   for (kk=0; kk<n; kk+=b)
4    for (i=ii; i<min(ii+b,n); i++)
5     for (j=jj; j<min(jj+b,n); j++)
6      for (k=kk; k<min(kk+b,n); k++)
7       C[i][j]+=A[i][k]*B[k][j];
```

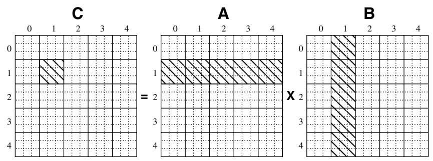


Fig. 7. Tile reduction: upper and lower bounds are functions

reduction tiles are not fixed values. In addition, the matrix `C[][]` might not be able to be evenly blocked. So, the tiles located at the margin of the matrix are usually smaller than the tiles located inside of the matrix. Thus, the sizes of the reduction tiles are not necessarily the same. All these information is reflected in the lower and upper bound expressions (or functions) in the dimension descriptor. Moreover, there is a restriction for the lower bound and upper bound expressions. They should not be functions of any index variable in the reduction kernel loops, i.e., they are orthogonal. This is to make sure that the shape of the reduction tile is a rectangle, or high-dimensional rectangle.

An interesting observation of this example code is that the number of the reduction kernel loops (which is 3, from line 4 to line 6) is not the same as the dimension of the reduction tile (which is 2). Generally, we do not require the number of the reduction kernel loops to be the same as the dimension of the reduction tile. We only require that the operations performed by the code in the reduction kernel loops can be viewed as one associative and commutative *macro* operation performed on the entire reduction tile.

3.2 Code Generation

Since tile reduction is derived from scalar reduction, its code generation shares the same framework as scalar reduction. Thus, we illustrate the code generation for tile reduction under the same framework as scalar reduction and use the code generation for scalar reduction as a reference. Generally, the code generation needs to deal with the following problems:

1. Distribute the iterations of the parallelized loop among the threads;
2. Allocate memory for the private copy of the tile used in the local recursive calculation;
3. Perform the local recursive calculation which is specified by the reduction kernel loops;
4. Update the global copy of the reduction tile;

Figure 8 shows the code generated for the tile reduction example in Figure 7. To make the paper easy to follow, we present the pseudo C code in the figure.

As we have mentioned at the beginning of Section 3.1, we try to avoid complicating the code generation when we were developing the extension for the `reduction` clause. A good example is the code generation for distributing the iterations of the parallelized loop among the dynamic threads. Actually, this part of the code generation for tile reduction is the same as that for scalar reduction.

```
0
1  /* statically partition the iteration space among the threads */
2  num_thr = __builtin_omp_get_num_threads ();
3  thr_id = __builtin_omp_get_thread_num ();
4  chunk_size = (((n+(b-1))/(b-1))%num_thr) == 0 ? \
     (((n+(b-1))/(b-1))/num_thr) : (((n+(b-1))/(b-1))/num_thr)+1;
5  lb = chunk_size * thr_id;        /* lower bound */
6  ub = min((lb+chunk_size),n); /* upper bound */
7
8  /* allocate memory for private tile */
9  private_tile = (int
*)__builtin_omp_memory_alloc( \
                 (min(ii+b,n)-ii)*(min(jj+b,n)-jj)*sizeof(int));
10
11  /* local tile reduction: serial */
12  for (kk=lb; kk<ub; kk+=b)
13   for (i=ii; i<min(ii+b,n), i++)
14     for (j=jj; j<min(jj+b,n), j++)
15       for (k=kk; k<min(kk+b,n), kk++)
16         private_tile[i-ii][j-jj] += A[i][k]*B[k][j]
17
18  /* update the global reduction tile */
19  __builtin_omp_atomic_start ();
20  for (i=ii; i<min(ii+b,n), i++)
21    for (j=jj; j<min(jj+b,n), j++)
22      C[i][j] += private_tile[i-ii][j-jj];
23 __builtin_omp_atomic_end ();
24
25  free(private_tile);
26
```

Fig. 8. Pseudo code generated for the matrix multiplication example to perform tile reduction

In the tile reduction program, the reduction kernel loops can be viewed as a single statement that performs the recursive calculation, which is the same as its counterpart in the scalar reduction program. So, from the angle of iteration distribution, the scalar reduction code and the tile reduction code are logically the same. Therefore, the method used to generate iteration distribution code for scalar reduction can also be used to generate iteration distribution code for tile reduction. It doesn't matter which `schedule` policy (`static`, `dynamic`, `guided`, or `runtime`) is deployed.

In Figure 8, we use the `static` scheduling policy as an example. In the code from line 2 to line 6, the iterations of the `kk` loop (line 3 in Figure 7) are evenly distributed among the threads. The iterations of the loop are divided into chunks and each chunk is assigned to one dynamic thread. The iteration chunk assigned to the thread is delimited by the lower bound variable `"lb"` and the upper bound variable `"ub"`, which are determined by the *thread number* of the owner thread. This piece of code only deals with the parallelized loop and the user specified OpenMP parameters. It does not even need to look into the code of the reduction kernel loops. This is the same for other schedule policies.

In line 9, the OpenMP runtime routine allocates memory for the the private tile (`private_tile`), which is a 2-dimensional array. This private tile is used by the thread as a temporary storage to perform the local sequential tile reduction. Its size is calculated from the parameters specified in the dimension descriptor (see equation 1). Its element data type is inferred from the tile name. All this information is obtained from the extended `reduction` clause.

The local sequential tile reduction is performed in the code from line 12 to line 16. This piece of code is almost the same copy as the original sequential program (line 3 to line 7 in Figure 7) except two places. In line 12, the lower and upper bounds of the loop are changed to `"lb"` and `"ub"`. This is to restrict the range of the iteration space in the chunk assigned to the current thread. Besides, in line 16, we replace the original reduction tile with the private tile and update its indices. This index calibration is required because the global reduction tile is cut out from a bigger multi-dimensional array, while the private tile is a standalone array. This piece of code performs local tile reduction sequentially, as in the original un-parallelized code.

After finishing the local tile reduction, the thread must update the global reduction tile. The code is shown in line 19 to line 23. The runtime routines in lines 19 & 23 ensure atomic access to the global reduction tile. The loops in line 20 and line 21 are extracted from the *reduction kernel loops*. Only the loops listed in the *tile descriptor* are selected. So, the loop `k` in the reduction kernel loops is not included. The *lhs* variable of the statement in line 22 is the same variable as in the original code (line 7 in Figure 7). However, the *rhs* variable has been replaced with the private tile, in which the indices have been updated.

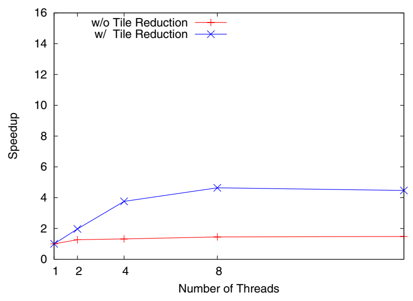
From the code in Figure 8, it is easy to see that the code generation for the tile reduction is as easy as that for the traditional scalar reduction. Meanwhile, no extra runtime supports is required. These advantages make the implementation of tile reduction in the OpenMP compiler very easy. In the next section, we will present the experimental results of applying the tile reduction on several typical benchmarks.

4 Experiments

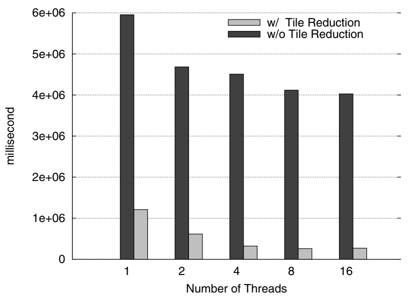
We have applied tile reduction on three benchmarks: the 2D histogram reduction, matrix-matrix multiplication and matrix-vector multiplication. The required code generation was implemented through source-to-source transformation and was prototyped in the Omni-1.6 OpenMP compiler [17]. The machine used in the experiments has 4 Intel Dual-Core Xeon (Paxville) chips, which are clocked at 3.0 GHz. Each core has HyperThreading (HT) enabled. Therefore, the machine can be viewed as a 16-processor shared memory parallel computer. Each chip has 4MB L2 cache (2MB each core) and each core has 16KB L1 cache.

Figure 9 shows the experimental data of the three benchmarks. The curve graphs on the left column display the speedup of the benchmark programs parallelized either through the tile reduction clause (w/ tile reduction) or through the standard OpenMP APIs (w/o tile reduction). The bar charts on the right column demonstrate the difference of the absolute execution time between the corresponding programs (w/ and w/o tile reduction) of the same set of benchmarks.

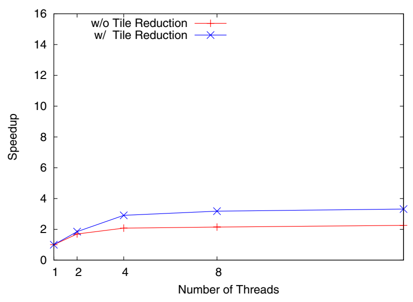
Figure 9(b) shows great performance enhancement if we parallelize the 2D histogram reduction benchmark with the tile reduction clause. Generally, compared with the program parallelized with standard OpenMP pragma, the absolute execution time of the tile reduction version decreased about 90% and its speedup on 8 threads increased from 1.5 to 4.5. The performance gain comes from the improved data locality, which owes to the tile reduction optimization. Without using tile reduction, the 2D histogram reduction



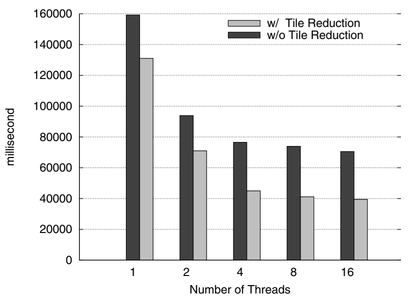
(a) 2D histogram reduction: speedup



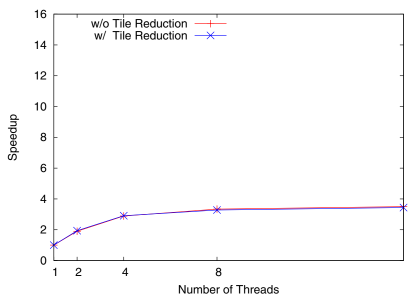
(b) 2D histogram reduction: execution time



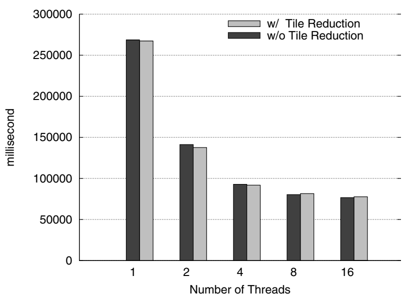
(c) Matrix-matrix multiplication: speedup



(d) Matrix-matrix multiplication: execution time



(e) Matrix-vector multiplication: speedup



(f) Matrix-vector multiplication: execution time

Fig. 9. Comparison of the speedup and execution time between the program parallelized with tile reduction and the program parallelized with the standard OpenMP pragma

program exhibit very poor scalability (shown in Figure 3). The tile reduction parallelization successfully rectifies the data access pattern and thus significantly improves its scalability. However, no matter what kind of optimizations are used, this benchmark stops scaling beyond 8 threads. This is because of the huge number of memory references in the code, which results in that its performance is finally restricted by the bandwidth of the shared memory bus.

The same phenomena are also observed in the matrix-matrix multiplication benchmark (see Figure 9(c) and 9(d)). Tile reduction can also decrease its execution time and improve its scalability. However, the magnitude of the performance enhancement caused by tile reduction is not as big as that of the 2D histogram reduction benchmark. This is also the same for the scalability enhancement. The reason is that the data locality of the tiled matrix-matrix multiplication program is better than the 2D histogram reduction benchmark. Therefore, the performance gain from tile reduction in the matrix multiplication program is less than that in the 2D histogram reduction program. On average, the execution time decreased 34% after applying tile reduction and its speedup increased from 2.15 to 3.18 on 8 threads and from 2.26 to 3.32 on 16 threads.

For the matrix-vector multiplication case, the performance enhancement brought about by tile reduction is smaller than that of the previous two benchmarks. The reason is the same as the previous one. Moreover, compared with the other two benchmarks, there are less data memory references in this benchmark. So, the program's performance degrades a little bit when it runs with 8 or 16 threads. This is because of the synchronization overhead caused by the code in line 19 and 23 in Figure 8. In average, its execution time decreased 0.28%.

5 Summary and Conclusions

In this paper, we introduced the concept of tile aware parallelization for OpenMP. Meanwhile, we developed the first tile aware parallelization technique - tile reduction, and illustrated the details of code generation for the tile reduction clause. We also designed a series of experiments to evaluate the tile reduction technique. From the experimental results and our experience of parallelizing the benchmarks, we have the following conclusions:

1. As a building block of the tile aware parallelization theory, tile reduction brings more opportunities to parallelize dense matrix applications.
2. For some benchmarks, tile aware parallelization is a more natural and intuitive way to reason about the best parallelization decision.
3. Tile reduction not only can improve data locality for some programs, but also can expose more parallelism.

6 Related Work

Parallel reduction operations are supported in many parallel programming languages. They include C**[18], SAC [19], ZPL [16], UPC [12], and MPI [13]. Most of them support user-defined reduction operations, either through language constructs or through library routines. User-defined reduction operation provides a flexible way to implement tile reduction. However, programmers need to change both data structures and algorithms, which, sometimes, is not a tirivial job.

Another piece of work that we need to mention is [20]. In [20], the authors propose to extend the OpenMP `reduction` clause to parallelize C++ generic algorithms. They propose to support user-defined types, overloaded operators, and function objects in the same way as the built-ins supported in the current OpenMP `reduction` clause.

Their work is very close to that presented in this paper. However, we study the reduction problem from a different angle. We propose tile reduction as one of the tile aware parallelizing technique for OpenMP, while [20] proposes user-defined reduction operation to complete their OpenMP extensions for parallelizing generic libraries. In our tile aware parallelization technique, we are concerned with the data partition, locality and a more flexible and efficient way to parallelize dense matrix programs written in cannonical C syntax, while the purpose of [20] is to allow people to parallelize programs written in modern C++ idioms such as *iterators* and *function objects*, which are not cannonical C syntax. Second, due to the non-trivial dynamic overhead of the generic techniques, generic libraries are not widely used in programming high performance scientific and engineering algorithms. Finally, there are no experimental data in [20].

7 Future Work

Tile reduction is one of the building block of the tile aware paralleization technique developed for OpenMP. One of our future work is to develop more parallelizing techniques (like tile reduction) such that OpenMP compiler can "recognize" data tiles and allow its runtime library to manipulate them. Our goal is to add tile aware parallelizing directives or clauses into the OpenMP programming interface. The purpose is to evolve OpenMP into an appropriate programming model for many-core processors with explicitly managed memory hierarchy [21], e.g. the IBM CELL [22] and the IBM Cyclops-64 [23] processor.

Acknowledgments

This work was supported by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors. We thank all the members of CAPSL group at University of Delaware. We thank Jason Lin and Lei Huang for their valuable comments and feedback.

References

1. Anderson, J.M., Amarasinghe, S.P., Lam, M.S.: Data and computation transformations for multiprocessors. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, Santa Barbara, California, July 19–21, pp. 166–178 (1995); SIGPLAN Notices 30(8) (August 1995)
2. Anderson, J.M., Lam, M.S.: Global optimizations for parallelism and locality on scalable parallel machines. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, Albuquerque, New Mexico, June 23–25, pp. 112–125 (1993); SIGPLAN Notices 28(6) (June 1993)
3. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, Toronto, Ontario, June 26–28, pp. 30–44 (1991); SIGPLAN Notices 26(6) (June 1991)

4. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Conference Record of POPL 1997: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, January 15–17, pp. 201–214 (1997)
5. High Performance Fortran Forum: High-performance fortran language specification version 2.0. Technical report, Rice University (1997)
6. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: Distributed Shared-Memory Programming. Wiley-Interscience, Hoboken (2003)
7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages and applications, pp. 519–538. ACM, New York (2005)
8. Deitz, S.J.: High-level programming language abstractions for advanced and dynamic parallel computations. Ph.D thesis, Seattle, WA, USA, Chair-Lawrence Snyder (2005)
9. Dotsenko, Y., Coarfa, C., Mellor-Crummey, J.: A multi-platform co-array fortran compiler. In: PACT 2004: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 29–40. IEEE Computer Society, Los Alamitos (2004)
10. Hilfinger, P.N., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G., Yelick, K.: Titanium language reference manual. Technical report, Berkeley, CA, USA (2001)
11. Guo, J., Bikshandi, G., Fraguela, B.B., Garzaran, M.J., Padua, D.: Programming with tiles. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 111–122. ACM, New York (2008)
12. UPC Consortium: UPC Collective Operations Specifications V1.0 A publication of the UPC Consortium (2003)
13. Forum, M.P.I.: MPI: A message-passing interface standard (version 1.0). Technical report (May 1994), `http://www.mcs.anl.gov/mpi/mpi-report.ps`
14. Deitz, S.J., Chamberlain, B.L., Choi, S.E., Snyder, L.: The design and implementation of a parallel array operator for the arbitrary remapping of data. In: PPoPP 2003: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 155–166. ACM, New York (2003)
15. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (May 2008), `http://www.openmp.org/mp-documents/spec30.pdf`
16. Deitz, S.J., Chamberlain, B.L., Snyder, L.: High-level language support for user-defined reductions. J. Supercomput. 23(1), 23–37 (2002)
17. Kusano, K., Satoh, S., Sato, M.: Performance evaluation of the omni openmp compiler. In: Valero, M., Joe, K., Kitsuregawa, M., Tanaka, H. (eds.) ISHPC 2000. LNCS, vol. 1940, pp. 403–414. Springer, Heidelberg (2000)
18. Viswanathan, G., Larus, J.R.: User-defined reductions for efficient communication in data-parallel languages. Technical Report 1293, University of Wisconsin-Madison (January 1996)
19. Scholz, S.B.: On defining application-specific high-level array operations by means of shape-invariant programming facilities. In: APL 1998: Proceedings of the APL 1998 conference on Array processing language, pp. 32–38. ACM, New York (1998)
20. Kambadur, P., Gregor, D., Lumsdaine, A.: Openmp extensions for generic libraries. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 123–133. Springer, Heidelberg (2008)
21. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: PPoPP 2007: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 226–236. ACM, New York (2007)

22. Eichenberger, A.E., O'Brien, K., O'Brien, K., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M.: Optimizing compiler for the cell processor. In: PACT 2005: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 161–172. IEEE Computer Society, Los Alamitos (2005)
23. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In: Workshop on Modeling, Benchmarking and Simulation (MoBS 2005) of ISCA 2005, Madison, Wisconsin (June 2005)

A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures

Eduard Ayguade[1,2], Rosa M. Badia[2], Daniel Cabrera[1], Alejandro Duran[1,2], Marc Gonzalez[1,2], Francisco Igual[3], Daniel Jimenez[1], Jesus Labarta[1,2], Xavier Martorell[1,2], Rafael Mayo[3], Josep M. Perez[2], and Enrique S. Quintana-Ortí[3]

[1] Universitat Politècnica de Catalunya (UPC)
[2] Barcelona Supercomputing Center (BSC-CNS)
[3] Universidad Jaume I, Castellon

Abstract. OpenMP has evolved recently towards expressing unstructured parallelism, targeting the parallelization of a broader range of applications in the current multicore era. Homogeneous multicore architectures from major vendors have become mainstream, but with clear indications that a better performance/power ratio can be achieved using more specialized hardware (accelerators), such as SSE-based units or GPUs, clearly deviating from the easy-to-understand shared-memory homogeneous architectures. This paper investigates if OpenMP could still survive in this new scenario and proposes a possible way to extend the current specification to reasonably integrate heterogeneity while preserving simplicity and portability. The paper leverages on a previous proposal that extended tasking with dependencies. The runtime is in charge of data movement, tasks scheduling based on these data dependencies and the appropriate selection of the target accelerator depending on system configuration and resource availability.

1 Introduction and Motivation

Computer architecture is under a revolution. The gigahertz race has stopped due to power dissipation problems. Fortunately, extra transistors continue being available in new chip generations, thanks to the technological reduction in transistor area. So new solutions are needed to use the extra transistors, and get lower power consumption at the same time. As a nice alternative to the gigahertz race, and instead of going for more complex pipelined and superscalar processors, the new transistors are used to incorporate functionalities that were external to the processor, into a single chip. For instance, GPUs, which are currently being used for general purpose computing, are becoming part of the processing chip [1,2]. Other approaches, like the Cell/B.E. processor [3] are targeting physics, encryption, and encoding. The purpose is to accelerate specific algorithms, so that applications can take advantage of the extra performance.

Future supercomputers will be equipped with heterogeneous hardware, including Cell processors, GPUs and even FPGAs. This hardware can provide dramatic performance advantages for high-performance computing applications, specially for applications featuring data-parallelism. In this scenario, programmers will have to deal with architectures composed by a mix of regular multicore CPUs and accelerators, possibly several types of accelerators, each one with its own programming environment and libraries, and possibly its own memory address space.

In order to overcome the programming challenges introduced by accelerator-based architectures, programming models need to evolve including features that allow to migrate applications to heterogenous architectures in a simple and portable way. If current application developers are still having a hard time trying to extract reasonable performance from homogeneous multicore architectures, the situation is about to get even worse with the emergence of heterogeneous multicore architectures.

The majority of proposals today assume a host-directed programming and execution model with attached accelerator devices. The bulk of a user application executes on the host while user-specified code regions are offloaded to the accelerator. In general, the specifics of the different accelerator architectures makes programming extremely difficult if one plans to use the vendor-provided SDKs (libspe for Cell, CUDA for Nvidia GPUs [4], ...). It would be desirable to retain most of the advantages of using these SDKs but in a much more accessible way, avoiding the mix of hardware specific code (for task offloading, data movement, ...) with application code.

In order to motivate the paper and show the complexities in using vendor-provided SDKs, we consider a simple matrix multiplication example in Figure 1. In this code, the programmer defines that each element of matrices `A`, `B` and `C` is a pointer to a block of `BS*BS` elements, which are allocated from inside the main function.

This simple example would require very different accelerator-dependent code in order to offload the execution, transfer data and synchronize host and accelerator(s). For example, to program an Nvidia GPU using CUDA, the programmer has to write the host and device codes. Figure 2 shows the code executed on the host, in which the programmer first needs to allocate memory on the GPU for the blocks of `A`, `B` and `C` (since the GPU executes from its own separate memory). Then the host copies matrices from host memory to GPU memory. The arguments give the destination pointer, the source pointer, the two dimension sizes, and copy direction. Then the host specifies the execution parameters and instantiates the kernel itself. Finally the host waits for the kernel to finish, moves results back from the GPU to the host and deallocates memory on the GPU.

For the Cell/B.E. the programmer would have to write two codes: one for the PPE (Figure 3) and one for the SPE (Figure 4). The PPE code handles thread allocation and resource management among the SPEs. In the example, the PPE code is creating a thread context, loading program, creates the thread and gets thread control for each of the SPEs. Then PPE code leaves the SPEs to perform the computation and afterwards waits for the SPE threads to finish. The SPE code iterates while there are matrix blocks to calculate. The function *next_block*

```
void matmul (float *A, float *B, float *C ) {
for (int i=0; i < BS; i++)
   for (int j=0; j < BS; j++)
      for (int k=0; k < BS; k++)
         C[i*BS+j] += A[i*BS+k] * B[k*BS+j];
}

float *A[NB][NB], *B[NB][NB], *C[NB][NB];

int main( void ){
int i, j, k;

for(i = 0; i < NB; i++)
    for(j = 0; j < NB; j++) {
        A[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
        B[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
        C[i][j] = ( float* ) malloc(BS*BS*sizeof(float));
    }

for (i = 0; i < NB; i++)
   for (j = 0; j < NB; j++)
      for (k = 0; k < NB; k++)
          matmul ( A[i][k], B[k][j], C[i][j] );
}
```

Fig. 1. Matrix multiplication example to motivate the extension of OpenMP for heterogeneous architectures

```
__global__ void matmul_kernel( float *A, float *B, float *C );

#define THREADS_PER_BLOCK 16

void matmul (float *A, float *B, float *C ) {
...
// allocate device memory
float *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, BS*BS*sizeof(float));
cudaMalloc((void**) &d_B, BS*BS*sizeof(float));
cudaMalloc((void**) &d_C, BS*BS*sizeof(float));

// copy host memory to device
cudaMemcpy(d_A, A, BS*BS*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, BS*BS*sizeof(float), cudaMemcpyHostToDevice);

// setup execution parameters
dim3 threads(THREADS_PER_BLOCK, THREADS_PER_BLOCK);
dim3 grid(BS/threads.x, BS/threads.y);

// execute the kernel
matmul_kernel<<< grid, threads >>>(d_A, d_B, d_C);

// copy result from device to host
cudaMemcpy(C, d_C, BS*BS*sizeof(float), cudaMemcpyDeviceToHost);

// clean up memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
}
```

Fig. 2. Matrix multiplication example (non optimized) targeting CUDA

```
void matmul_spe ( float *A, float *B, float *C );

void matmul ( float *A, float *B, float *C ) {
for (i=0; i<num_spus; i++) {
    // Initialize the thread structure and its parameters
    ...
    // Create context
    threads[i].id = spe_context_create (SPE_MAP_PS, NULL);
    // Load program
    rc = spe_program_load (threads[i].id, &matmul_spe)) != 0;
    // Create thread
    rc = pthread_create (&threads[i].pthread, NULL,
                          &ppu_pthread_function, &threads[i].id);
    // Get thread control
    threads[i].ctl_area = (spe_spu_control_area_t *)
                          spe_ps_area_get(threads[i].id, SPE_CONTROL_AREA);
    }
// Start SPUs
for (i=0; i<spus; i++) send_mail(i, 1);
// Wait for the SPUs to complete
for (i=0; i<spus; i++)
    rc = pthread_join (threads[i].pthread, NULL);
}
```

Fig. 3. Matrix multiplication example (non optimized and omitting conditional statements to control error codes) for Cell/B.E. using IBM's SDK: PPE side

```
void matmul_spe ( float *A, float *B, float *C )
{
...
   while (blocks_to_process()){
      next_block(i, j, k);
      calculate_address (baseA, A, i, k);
      calculate_address (baseB, B, k,j);
      calculate_address (baseC, C, i, j);
      mfc_get(localA, baseA, sizeof(float)*BS*BS,in_tags, 0, 0);
      mfc_get(localB, baseB, sizeof(float)*BS*BS,in_tags, 0, 0);
      mfc_get(localC, baseC, sizeof(float)*BS*BS,in_tags, 0, 0);
      mfc_write_tag_mask((1<<(in_tags)));
      mfc_read_tag_status_all();     /* Wait for input data
      for (ii=0; ii < BS; ii++)
         for (jj=0; jj < BS; jj++)
            for (kk=0; kk < BS; kk++)
               localC[i][j]+= localA[i][k]* localB[k][j];
      mfc_put(localC, baseC, sizeof(float)*BS*BS, out_tags, 0, 0);
      mfc_write_tag_mask((1<<(out_tags)));
      mfc_read_tag_status_all();     /* Wait for output data
      }
...
}
```

Fig. 4. Matrix multiplication example for Cell/B.E. (non optimized) using IBM's SDK: SPE side

performs atomic counter increments between all SPEs. The SPE code performs three DMA read operations (for blocks from matrices A, B and C) and blocks until the transfers have finished. The block matrix operation is then performed locally and finally a DMA write operation is performed for the block from matrix C. This code is quite naive, since for example no double buffering nor SIMDization

```
void matmul (float *A, float *B, float *C ) {
// configure device
int al_desc = rasclib_algorithm_open("matrixmult");

// queues up command to send inputs
res = rasclib_algorithm_send(al_desc , "a", A, BS*BS*sizeof(float));
res = rasclib_algorithm_send(al_desc , "b", B, BS*BS*sizeof(float));
res = rasclib_algorithm_send(al_desc , "c", C, BS*BS*sizeof(float));

// queues up command to execute bitstream
rasclib_algorithm_go(al_desc);

// queues up command to receive results
res = rasclib_algorithm_receive(al_desc , "c", C, BS*BS*sizeof(float));

// wait for termination
rasclib_algorithm_committ(al_desc , NULL);
rasclib_algorithm_wait(al_desc);
}
```

Fig. 5. Matrix multiplication example (non optimized and omitting conditional statements to control error codes) targeting a RASC Altix blade

are used to optimize the code. The sample matrix multiply code from the IBM SDK has more than 600 lines for the PPE code and more than 1300 lines for the SPE code.

For a system like the SGI Altix with a Reconfigurable Application Specific Computing (RASC) FPGA blade, the code in Figure 5 using calls to the RASClib library would be used (in addition to generate the bitstream `matrixmult` with the FPGA compiler). The code assumes that the application has already reserved and configured the FPGA device. The `rasclib_algorithm_open` allocates all necessary internal data structures for a logical algorithm. The three `rasclib_algorithm_send` calls pull data down to the input data areas on the device. The `rasclib_algorithm_go` starts execution. The `rasclib_algorithm_receive` call pushes the result back out to host memory. The `rasclib_algorithm_commit` causes all of the commands that have been queued up by the previous calls to be sent to the device. The `rasclib_algorithm_wait` blocks until all the command that were sent to all of the devices are complete, then returns.

There have been substantial efforts to propose and develop programming models for hybrid architectures that abstract the target architecture. These differ in the objects they manipulate, in general arrays or matrices. For example both RapidMind [5] and PeakStream are stream languages that operate on streams (vectors of arbitrary length) of data embedded in C or C++ and they are oriented to GPGPUs. For the ClearSpeed floating-point accelerator, the C^n programming language adds new datatypes (mono and poly) to indicate if there is only one instance of the data or all functional units have a portion of the data. Others (e.g. PGI) propose directives to delineate accelerator regions and extract parallelism in loops or follow a task parallelism model, and offer architecture independent abstractions for offloadable functions (e.g. Sequoia [6], Merge [7], CellSs [8], HMPP [9]). A growing number of OpenMP compiler frameworks are also intended to offer support for heterogeneous architectures (e.g. Octopiler [10] for Cell or PGI [11] and [12] for CUDA).

Most of the proposals and environments available target a single type of accelerator at a time. In order to have the same code prepared to compile for several target accelerators, the programmer needs to use conditional compilation to isolate declaration of variables and calls to different APIs for the different target devices.

2 Proposed OpenMP Extensions

In this section we propose a set of extensions to OpenMP 3.0 to express the execution of tasks on a hardware accelerator. This extension leverages on a previous proposal to allow the specification of dependencies between tasks [13], although it can be considered totally independent.

Tasks are the most important new feature of OpenMP 3.0. A programmer can define deferrable units of work, called tasks, and later ensure that all the tasks defined up to some point have finished.

```
#pragma omp task [clause-list]
   structured-block
```

Valid clauses are `shared`, `private`, `firstprivate` and `untied`. The first three are used for setting data sharing attributes of variables in the task body; the last one specifies that the task can be resumed by a different thread after a possible *task switching point*. The proposal in [13] extended the `task` construct with some additional clauses that are used to derive dependencies among tasks at runtime: `input`, `output` and `inout`. Although in some cases the compiler can analyze the code and determine the input and output data sets, we provided these additional clauses to modify or augment the compiler analysis.

OpenMP allows the specification of any structured block inside the `task` construct. This motivates the presentation of our proposal in two parts. The first part of our proposal just allows the specification of target devices for the execution of a task. In the second part, we consider a subset of the possible tasks than can be expressed in OpenMP: tasks composed of a function call. In this case, the programmer will be able to specify alternative implementations for different target devices. For the general case we have not found a portable way to specify alternative implementations (each one targeting an accelerator device) for structured blocks of code.

2.1 Specifying Target Devices

Our proposal consists of a new pragma that may precede an existing pragma `task`:

```
#pragma omp target device(device-name-list) [clause-list]
```

The `target` construct specifies that the execution of the task could be offloaded on any of the devices specified in `device-name-list` (and as such its code must be handled by the proper compiler backends). If the task is not preceded by a `target` directive, then the default `device-name`, which is `smp` and corresponds to a homogeneous shared-memory multicore architecture, will be used. Other `device-names` are vendor specific (we will use along this paper three possible examples: `cell`, `cuda` and `fpga`). When a task is ready for execution (i.e. it has no dependencies with other previously generated tasks) the runtime can choose among the different available targets to decide in which device to execute the task. This decision is implementation-dependent but it will ideally be tailored to resource availability. If no resource is available, the runtime will stall that task until one becomes available.

Some restrictions may apply to tasks that target a specific device (for example, they may not contain any other OpenMP directives, do any input/output, ...). In addition, tasks offloaded in some specific devices should be tied or they should execute in the same type of device if thread switching is allowed.

Some additional clauses can be used with this pragma `device`:

– `copy_in(data-reference-list)`
– `copy_out(data-reference-list)`

These two clauses, which are ignored for the `smp` device, specify data movement for `shared` variables used inside the task. `Copy_in` will move variables in `data-reference-list` from host to device memory. `Copy_out` will move variable in `data-reference-list` back from device to host memory. Once the task is ready for execution, the runtime system will move variables in the `copy_in` list. Once the task finishes execution, the runtime will move variables in the `copy_out` list, if necessary.

A *data-reference* in a *data-reference-list* can contain a variable identifier or a reference to subobjects. References to subobjects include array element references (like `a[4]`), array sections (like `a[3:6]`), field references (like `a.b`) and shaping expressions (like `[10][20] p`). Since C does not have any way to express ranges of an array, we have borrowed the *array-section* syntax from Fortran 90. These array sections, with syntax `a[e1:e2]`, designate all elements from `a[e1]` to `a[e2]` (both ends are included and `e1` shall yield a lower or equal value than `e2`). Multidimensional arrays are eligible for multidimensional array sections (like `a[1:2][3:4]`). While not technically naming a subobject, non-multidimensional array section syntax can also be applied to pointers (i.e.: `pA[1:2]` is valid for `int *pA`, but note that `pB[1:2][3:4]` is invalid for `int **pB`, also note that `pC[1:2][3:4]` is valid for `int *a[N]` and so it is `pD[1:2][3:4][5:6]` for `int *a[N][M]`). For syntactic economy `a[:x]` is the same as `a[0:x]` and, only for arrays where the upper bound is known, `a[x:]` and `a[:]` mean respectively `a[x:N]` and `a[0:N]`. Designating an array (i.e.: `a`) in a data reference list, with no array section nor array subscript, is equivalent to the whole array-section (i.e.: `a[:]`). Shaping expressions are a sequence of dimensions, enclosed in square brackets, and a data reference, that should refer to a pointer type (like `[10][20] p`). These shaping expressions are aimed at those scenarios where an array-like structure has been allocated but only a pointer to its initial element is available. The goal of shaping expressions is to provide to the compiler such unavailable structural information.

Other vendor-specific clauses in the `target` construct for each particular `device-name` are possible.

2.2 *Taskifying* Functions

Our second proposal applies to those tasks that are just composed of a function call

```
#pragma omp task [clause-list]
   function-call
```

In this paper we also consider another way to specify tasks in OpenMP, which we have found very convenient to *taskify* functions that are always executed as tasks:

```
#pragma omp task [clause-list]
   {function-header|function-definition}
```

Whenever the program calls a function annotated in this way, the runtime will create an explicit task.

In this case, the pragma proposed in the previous subsection applies to a function header or definition:

```
#pragma omp target device(device-name-list) [clause-list]
   {function-header|function-definition}
```

The `target` construct specifies that the function contains code prepared for its execution on all devices specified in `device-name-list`. If a function is not preceded by a `target` directive, then the default `smp` device is used. In addition to the possible clauses specified in the previous section, we allow in this case the following one:

– `implements(function-name)`

This clause `implements` is used to specify an alternative implementation for a function. For example:

```
#pragma omp task
void matmul( float *A, float *B, float *C );
...
#pragma omp target device(cell) implements(matmul)
void matmul_cell( float *A, float *B, float *C ) {
... // optimized version for target device
}
```

or directly in the header of a routine in an optimized library:

```
#pragma omp task
void matmul( float *A, float *B, float *C );
...
#pragma omp target device(cell) implements(matmul)
void matmul_spe( float *A, float *B, float *C );
```

The programmer is specifying that the alternative function (matmul_cell or matmul_spe) should be used instead of function matmul when offloading the task to one of the Cell SPE units. If the device cell is not available, then the runtime will launch the execution of the original matmul function on the default smp device. Different names are used for the different implementations in order to avoid duplicated symbols.

If the original implementation is appropriate for one of the accelerator types, then the programmer should precede the definition of the task with the specification of the target device

```
#pragma omp target device(smp,cell)
#pragma omp task
void matmul ( float *A, float *B, float *C ) {
... // original sequential code
}
```

In this case, the compiler would generate two versions for the same function, one going through the native optimizer for the default device and another going through the accelerator-specific compiler.

2.3 A Couple of Examples

Figure 6 shows the same matrix multiplication example used in section 1. The programmer specifies in the code example that the task could be offloaded into one of the Cell SPEs. The code to be offloaded should be generated by the native compiler for Cell using the function definition in matmul. Note that the inout clause [13] is used in the definition of the task to express the data dependence that exists among tasks computing the same block of C.

Several target accelerator devices can be specified in the application. For example in Figure 7 the programmer is specifying three possible options to execute function matmul. The first one is using the original definition of function matmul for the default target architecture. Two alternatives specified by the user are the implementation specified in matmul_cuda for an Nvidia GPU or the library implementation named matmul_spe for the IBM Cell. For all the devices, the runtime is in charge of moving data before and after the execution of the task.

2.4 Changing Memory Association for Data

In some kind of accelerators, as for instance in the Cell SPUs or in GPUs, it may be necessary to change the memory association for the data in memory prior to the execution on the device. Our proposal based on alternative implementations

```
#pragma omp target device(smp,cell) copy_in(A[BS][BS], B[BS][BS], C[BS][BS])
                                    copy_out(C[BS][BS])
#pragma omp task inout(C[BS][BS])
void matmul( float *A, float *B, float *C ) {
  // original sequential code in Figure 1
}

float *A[NB][NB], *B[NB][NB], *C[NB][NB];

int main( void ){
for (int i = 0; i < NB; i++)
   for (int j = 0; j < NB; j++)
      for (int k = 0; k < NB; k++)
         matmul ( A[i][k], B[k][j], C[i][j] );
}
```

Fig. 6. Example specifying the execution on a device

```
#pragma omp task inout(C[BS][BS])
void matmul( float *A, float *B, float *C) {
  // original sequential code in Figure 1
}

#pragma omp target device(cuda) implements(matmul)
            copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
void matmul_cuda ( float *A, float *B, float *C) {
  // optimized kernel for cuda
}

#pragma omp target device(cell) implements(matmul)
            copy_in(A[BS][BS], B[BS][BS], C[BS][BS]) copy_out(C[BS][BS])
void matmul_spe ( float *A, float *B, float *C);

float *A[NB][NB], *B[NB][NB], *C[NB][NB];

int main( void ){
for (int i = 0; i < NB; i++)
    for (int j = 0; j < NB; j++)
        for (int k = 0; k < NB; k++)
        matmul (A[i][k], B[k][j], C[i][j]);
}
```

Fig. 7. Example with the specification of alternative implementations for several target devices

could allow a runtime to change of memory association by specifying different headers for the implemented functions.

For example consider the code in Figure 8 that uses contiguous storage for matrices `A`, `B` and `C`. Notice that `matmul` defines the arguments as `[N][N]` and only accesses blocks of size `[BS][BS]` while `matmul_block` defines the arguments as `[BS][BS]`. The compiler can recognize this and instruct the runtime system to do the data movement to the local memory of the Cell SPE in a blocked way.

The extensions proposed in this paper are orthogonal to other possible extensions to generate efficient code by a compiler (e.g., vectorization width, number of threads running on accelerators, code transformations, ...) could be necessary for the compiler to generate good code for the target device. Proposals commented in the next section address some of these issues.

```
#pragma omp task inout(C[BS][BS])
void matmul (float A[N][N], float B[N][N], float C[N][N] ) {
  // original sequential code in Figure 1
}

#pragma omp target device(cell) implements(matmul) copy_in(A, B, C) copy_out(C)
void matmul_block( float A[BS][BS], float B[BS][BS], float C[BS][BS]) {
  // original sequential code in Figure 1
}

float A[N][N], B[N][N], C[N][N];

int main( void ){
for (int i = 0; i < N; i=i+BS)
   for (int j = 0; j < N; j=j+BS)
      for (int k = 0; k < N; k=k+BS)
      matmul ( &A[i][k], &B[k][j], &C[i][j] );
}
```

Fig. 8. Example with the specification of an implementations with change of memory association

3 Related Work

In this section we focuss on two approaches more closely related to our proposal (HMPP [9] and PGI [11]). We summarize both proposals in terms of specification of code regions to be executed on accelerator devices and how/when they decide to offload the execution.

3.1 CAPS Hybrid Multi-core Parallel Programming (HMPP)

HMPP [9] is designed to simplify the use of accelerators while keeping the application code portable (a sequential binary version can be built using a traditional compiler). The HMPP approach is to declare, by means of directives, functions (named codelets) suitable for hardware acceleration and callsites to them. Codelets are pure functions (i.e. functions that always evaluate the same result value given the same argument value(s), and have no side effects and no I/O). The accelerator functions are written in the own accelerator language in a specific file while keeping the original computation in the main source; then the developer uses the accelerator specific provided tools (compiler, library, ...) to generate the function binary.

The general syntax of the HMPP pragmas

```
#pragma hmpp <label> <directive-type> [, <directive-parameter>]*
```

The main `directive-types` are `codelet` (allows the declaration of a function as a codelet) and `callsite` (allows the call of a codelet). `label` is a unique identifier for a couple (codelet, callsite). The `directive-parameters` also specify the accelerator target (e.g. `target=cuda:sse`), conditional execution of the codelets (e.g. `cond=expr("n==1024")`, their desired synchronous or asynchronous properties and the data transfers (`input`, `output` and `inout` followed by the name of a function argument). In order to support these data transfers, there are constraints on how the codelet arguments are specified (the argument coding rules have to permit to compute the amount of data to transfer runtime). For example, for n-dimensional data structures, the argument is followed by a one-dimensional array argument whose n elements give the size of each dimension.

The `synchronize` directive types allow to wait for the termination of a codelet. Other directive types are for decoupling the data transfers from the computations. By preloading (`advanceload`) data and downloading the results (`delegatedstore`) whenever they are required in the main application, the programmer can optimize the use of the memory bandwidth. Programmer can use the `asynchronous` directive parameter to interlace data transfers and codelet execution or the `const` directive parameter to preload data only once.

At execution, the HMPP runtime takes care of discovering the attached accelerators and their availability. When a codelet is indicated to be run on an accelerator, if the device is available and if the shared library corresponding codelet is present, HMPP loads it just as a software plug-in. Otherwise the native version is run on the host CPU or in a worker thread. The use of dynamic linking in HMPP allows to add improved codelet versions or add codelets for new hardware accelerators, without recompiling the overall application source.

The HMPP approach is quite similar to the proposal in this paper, since it also annotates functions to be offloaded in the accelerators that are specified in the `target` clause. We think that our approach has a better potential to express multiple implementations of functions, it is better integrated in the OpenMP specification and makes programming easier by delegating intelligence to the runtime system.

3.2 PGI Directives and Intrinsic Functions

The directives and programming model defined by PGI [11] allow programmers to specify the regions of a host program to be targeted for offloading to an accelerator device (mainly GPUs), without the need to explicitly initialize the accelerator and manage data or program transfers between the host and accelerator. Rather, all of these details are implicit in the programming model and are managed by their accelerator compilers. The bulk of a user's program are executed on the host. Their current version does not support multiple accelerators of the same type or different types.

The proposed directives are used to: 1) delineate accelerator regions and 2) augment information available to the compiler for scheduling of loops. `#pragma acc region` defines the region of the program in which loops will be compiled into accelerator kernels. It accepts clauses to specify data that needs to be copied from the host memory to the accelerator memory (`copyin`) and result data that needs to be copied back (`copyout`). The `local` clause is used to declare that the data needs to be allocated in the accelerator memory. The programmer can use the `if(condition)` clause to instruct the compiler to generate two copies of the region, one copy to execute on the accelerator and one copy to execute on the host, and decide which one to execute based on the evaluation of `condition`. The accelerator loop mapping directive `#pragma acc for` applies to loops. It can describe what type of parallelism to use to execute the loop (`host [(width)]`, `parallel [(width)]`, `seq [(width)]` and `vector [(width)]`). If more than one scheduling clause appears on the loop directive, the compiler will strip-mine the loop to get at least that many nested loops, applying one loop scheduling clause to each level. The pragma also allows to declare loop `private` and `cache` variables, arrays and subarrays.

In summary, the main differences with our proposal is that PGI is based on compiler technology to optimize the offloading of loops in accelerators. Also the data movement between the memories is managed by the compiler, not by the runtime system.

4 Conclusions and Future Work

This paper proposes an extension to the OpenMP 3.0 tasking model to reasonably integrate heterogeneity while preserving simplicity and portability. Our proposal allows the programmer to easily specify the target devices for the execution of a task as well as, for a subset of the tasks that can be expressed in OpenMP, alternative implementations of the task for different target devices. We have shown how with this proposal the programmer could extend a simple matrix multiply to specify the execution in an heterogenous environment.

An implementation of this proposal is currently undergoing. We are currently trying accommodate in our extension other proposals targeting streaming architectures, such as the one proposed in [14], in which tasks become stream filters and `copy_in` and `copy_out` clauses are used to indicate input and output streams. Finally, we are also investigating new pragmas to direct program transformation (for instance to specify loop blocking) and their interaction with OpenMP constructs and our proposed extensions.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the SARC project (contract no. 27648) and the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. The researchers at the Universidad Jaime I were supported by the Spanish Ministry of Science and Innovation/FEDER (contracts no. TIN2005-09037-C02-02 and TIN2008-06570-C04-01) and by the *Fundación Caixa-Castellón/Bancaixa* (contracts no. P1B-2007-19 and P1B-2007-32).

References

1. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. 27(3), 1–15 (2008)

2. AMD Corporation. AMD Fussion
3. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., et al.: The Design and Implementation of a First-Generation Cell Processor. In: IEEE International Solid-State Circuits Conference, ISSCC 2005 (2005)
4. NVIDIA corporation. NVIDIA CUDA Compute Unified Device Architecture Version 2.0 (2008)
5. RapidMind. RapidMind Multi-core Development Platform, `http://www.rapidmind.com/pdfs/RapidmindDatasheet.pdf`
6. Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J., Hanrahan, P.: Compilation for explicitly managed memory hierarchies. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007)
7. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 287–296 (2008)
8. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (2006)
9. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: First Workshop on General Purpose Processing on Graphics Processing Units (October 2007)
10. O'brien, K., O'brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting OpenMP on Cell. International Workshop on OpenMP 2007, International Journal of Parallel Programming 36(3), 289–311 (2008)
11. The Portland Group. PGI Fortran and C Accelerator Compilers and Programming Model Technology Preview
12. Lee, S., Min, S.-J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Symposium on Principles and Practice of Parallel Programming (PPoPP 2009) (February 2009)
13. Duran, A., Pérez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP tasking model to allow dependent tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
14. Martorell, X., Ramirez, A., Carpenter, P., Rodenas, D., Ayguade, E.: Streaming machine description and programming model. In: Proceedings of the 7th International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS), pp. 107–116 (2007)

Identifying Inter-task Communication in Shared Memory Programming Models

Per Larsen, Sven Karlsson, and Jan Madsen

DTU Informatics
Technical University of Denmark
{pl,ska,jan}@imm.dtu.dk

Abstract. Modern computers often use multi-core architectures, covering clusters of homogeneous cores for high performance computing, to heterogeneous architectures typically found in embedded systems. To efficiently program such architectures, it is important to be able to partition and map programs onto the cores of the architecture. We believe that communication patterns need to become explicit in the source code to make it easier to analyze and partition parallel programs. Extraction of these patterns are difficult to automate due to limitations in compiler techniques when determining the effects of pointers.

In this paper, we propose an OpenMP extension which allows programmers to explicitly declare the pointer based data-sharing between coarse-grain program parts. We present a dependency directive, expressing the input and output relation between program parts and pointers to shared data, as well as a set of runtime operations which are necessary to enforce declarations made by the programmer. The cost and scalability of the runtime operations are evaluated using micro-benchmarks and a benchmark from the NAS parallel benchmark suite. The measurements show that the overhead of the runtime operations is small. In fact, no performance degradation is found when using the runtime operations in the benchmark from the NAS parallel benchmark suite.

1 Introduction

The adoption of parallel programming is starting to reach outside the parallel and scientific computing communities and the use of multi-core processors in desktop and embedded systems calls for parallel programming in all application areas. Parallel programming requires the programmer to partition and map the program onto the different processing elements.

A key challenge in partitioning a program is determining the communication between coarse-grain program parts also known as *tasks*. In shared memory programming models, communication is done through memory which means that it is important to determine the effects of pointers. However, analysis techniques to determine the effects at compile time produce over-approximations leading to pessimistic choices in the subsequent scheduling and mapping steps. The alternative, which is to determine dependencies by hand, is time-consuming and error prone.

Due to the relaxed-consistency shared memory model used in OpenMP [1, p. 21], a conforming program can be viewed as being partitioned, by synchronization operations, into a set of tasks. Programmer declarations of each task's access to shared data through pointers are used, rather than program analysis, to determine data dependencies between tasks. As programmers may make errors, we do not assume that the program is correct with respect to the declarations. Instead, run-time checks are used to verify that the program conform to the communication patterns described in the directives.

This paper seeks to answer the following questions: i) can the programmer mitigate the limitations of analysis of pointer effects by explicitly declaring dependencies between tasks? ii) if so, can the correctness of the task dependency declarations be checked dynamically with a negligible run-time overhead?

In this paper, we will allow ourself to pessimistically overestimate the communication between tasks in two cases. First, data decomposition of arrays is not taken into account. Instead, we rather assume the entire array to be shared. Secondly, we assume that shared data structures are partially updated and ignore the possible optimizations that can be done if a data structure is written in its entirely by a task. Finally, we do not discuss the use of type systems for data sharing – various approaches have been compared by Liblit et al. [2].

The main contributions in this paper are the following. We propose an extension to OpenMP that, using directives, describes the data sharing patterns and hence the communication patterns in applications. We also describe the runtime support needed so that object code generated by a compiler can check at runtime that the program match the programmer declarations. Finally, we measure the overhead of the proposed extension using a kernel from the NAS parallel benchmarks [3,4] and in more detail using a set of micro-benchmarks. The measurements show that the run-time overhead of checking the declarations of a single task is approximately 2 microseconds whereas the cost of executing a single OpenMP `parallel` section is 42 microseconds using 8 threads. Also, no difference in execution time was observed between the original and instrumented versions of the benchmark kernel when measured with the GNU `time` utility.

The remainder of the paper is structured as follows. After a brief note on terminology, the proposed task dependency directive is introduced in Sect. 2. Section 3 discusses the background of our work and related work. Section 4 illustrates the pointer aliasing problem and discusses how it can be mitigated with the proposed task dependency declarations. Implementation of the runtime support is outlined in Sect. 5 after which the performance results are given in Sect. 6. Section 7 concludes the paper.

2 Task Dependency Declarations

Performance enhancing features of modern computer architectures as well as compiler optimizations may lead to overlapping as well as reordering of memory operations [5]. This also applies to OpenMP where each thread has a temporary view of memory that is not necessarily consistent with the main memory between memory synchronization operations [1]. A thread's view of memory is made consistent with the main memory by a memory synchronization operation or, in OpenMP terminology, a `flush`. Apart from `flush` operations performed explicitly by the programmer, many OpenMP sections imply a `flush` operation on entry and/or exit as will be explained shortly.

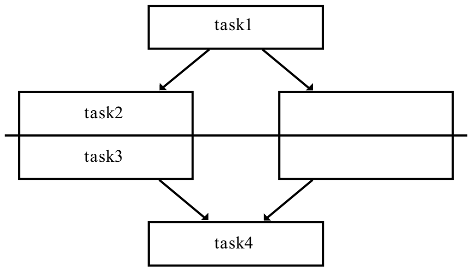
Due to the relaxed-consistency memory model, a properly written OpenMP program can not rely on writes made by one thread to be visible to another thread before both threads have executed a `flush`. Consequently, such programs can be viewed as being partitioned into a set of of tasks, each of which is a sequence of instructions delimited by `flush` operations. Conceptually, a task can not execute before all its inputs are available and no outputs are available before its computation has finished.

This definition is compatible with the general tasking concept found in the task scheduling literature [6] but differs from the notions of a task used to support irregular parallelism in OpenMP 3.0 [1] and SmpSs [7]. There, a task refers specifically to a sequence of instructions enclosed by an `omp task` or a `smpss task` directive, respectively.

The following definitions are used in this paper:

Task. A sequence of instructions delimited by any OpenMP directive or routine which includes, either implicitly or explicitly, a `flush` operation [1, p. 72]. All instructions belonging to the same task are executed in sequence.
Task-shared object. A C object which is accessed within more than one task. If the object is a structure, its fields are also task-shared.
Task-private object. A C object which is not task-shared.
Runtime-checked pointer. A pointer which may point to task-shared objects which is therefore subject to runtime checks as described in Sect. 2.

Two examples are given in Fig. 1. We use C as the base language but our work can be applied to all languages currently supported by OpenMP.



```
1 int *x = &arr
2 x[i] = ...
-----------------
3 int z[8];
4 z[j] = x[j]
```

a b

Fig. 1. a) Structure of a program using OpenMP `parallel` and `barrier` directives to fork, synchronize and join the execution of four tasks. b) Two snippets of code which belong to separate tasks. The pointer `x` is accessed directly on lines 1,2 and 4, `arr` is accessed via `x` on lines 2 and 4 and both are therefore task-shared whereas `z` is task-private – in addition `x` is a checked pointer as its pointee is task-shared.

2.1 The `depends` Directive

Inspired by the declaration of data-dependencies in Jade [8] and SmpSs [7], we propose an extension such that a task must declare not only which pointers provide input but also the tasks that produced the input. Similarly, a task that produces output through pointers must declare which tasks can consume the output.

The ability to precisely determine which tasks communicate through pointers via the proposed declarations depends on the correctness of the declarations. Therefore, it must be possible to detect whenever the declarations made by the programmer are incorrect. Verifying the correctness of the declarations at compile time is infeasible as we will explain in Sect. 4. Simple cases may be handled statically, but in the remaining cases the declarations must be enforced via runtime checks.

To enable efficient runtime checks, certain restrictions are put upon checked pointers. As they are compared by their value, checked pointers should always hold the address of the first element in case the pointee is of an array or aggregate type. For this reason pointer arithmetic is not allowed on runtime checked pointers.

The `depends` directive must be used to declare dependencies with other tasks through shared data. It must be placed immediately before the *task boundary* (defined below) which begins the task to which the declarations applies.

The syntax of a task dependency directive is given below using the informal notation of the OpenMP 3.0 specification [1].

`#pragma depends (`*name*`)` *[clause[[,] clause]...] new-line*
  task-boundary

where a valid *task-boundary* is at one of the following memory synchronization points as defined by the OpenMP specification [1, p. 291]

- a `barrier`.
- a `flush` with an empty flush-set.
- entry to or exit from a `parallel`, `critical` or `ordered` region.
- exit from a worksharing region unless a `nowait` clause is present.
- entry to or exit from a combined parallel worksharing region.
- immediately before or after an `omp task` scheduling point.
- entry to or exit from a `omp_set_lock` and `omp_unset_lock` region.

and *name* assigns a name to the task and *clause* is one of

`input` (*rt-checked-ptr, task-name[,task-name]...*)
`output`(*rt-checked-ptr, task-name[,task-name]...*)
`inout` (*rt-checked-ptr, task-name[,task-name]...*)

The `input` clause requires two types of arguments – *rt-checked-ptr* is a runtime checked pointer and each *task-name* is the name of a task that may have modified the pointee of *rt-checked-ptr* prior to the execution of the task which the `input` clause applies to. The arguments of the `output` clause is similar except that each *task-name* identifies a task that may read the pointee of *rt-checked-ptr* after the execution of the task which the `output` clause applies to.

The `inout` clause is a syntactic shorthand and `inout(ptr,t1,t2)` is equivalent to `input(ptr,t1,t2), output(ptr,t1,t2)`.

Any checked pointer which is accessed by a task must appear in the `depends` directive of that task. A pointer which is assigned to the value of a checked pointer is also checked cf. our definition.

A task t is only allowed to read a task-shared object pointed to by p when all of the following criteria are met:

– the pointer p appears in an `input` or `inout` clause in the `depends` statement of t; and
– any task t_{wr}, that writes to the task-shared pointee of p and can execute before t, appears in an `output` or `inout` clause in the `depends` directive of t. If control-flow analysis [9] can not determine that t_{wr} may only execute after t it must appear in the `depends` directive of t.

The conditions for dereferencing a checked pointer for writing are analogous.

To detect when the above criteria are violated, the value held in the checked pointer, i.e. the address of the task-shared object, is used as the key in a permission table which is initialized from the `depends` statements of an annotated program. As previously mentioned, a checked pointer must therefore point to the beginning of the shared object and thus we have chosen not to permit pointer arithmetic on checked pointers. An alternative solution would be to represent checked pointers as two regular pointers - one which holds first address of the task-shared object and one which can point to any part of the object. However, the use of such pointers to shared objects doubles the storage needed to hold a checked pointer and conversions may be required when interoperating with program libraries.

The proposed directives are assumed to be processed by a compiler. In this paper, we only translate the directives into calls to runtime functions by hand to provide a proof of concept. However, we have completed a full implementation of the necessary run-time functionality.

Section 5 will provide more insight on how to detect undeclared accesses to shared objects.

3 Background and Related Work

One of the most central aspects in parallel computing is the communication between parallel entities. We argue that i) the coarse-grain communication patterns in parallel code have to be explicit and ii) having such information available at compile time is of value as outlined in the following sections.

3.1 Mapping Applications to Heterogeneous Platforms

A task graph is a graph model that abstracts the performance and resource characteristics of a single application or workload. Task graphs are used in many contexts including system-level design where an optimal mapping of an application to a heterogeneous, application specific platform is sought after [10]. Premised on a survey of well known graph models for representation of parallel computation, Sinnen and Sousa [11] define a general graph model in which computation is associated with the vertices, called *nodes*, and communication is associated with its edges. One type of node is a task as defined in Sect. 2.

To be useful, the nodes and edges of a graph model must represent the computation and communication, respectively, of the actual application as precisely as possible. Approaches to graph model generation include extraction from source code by hand [10], via compiler analysis and instrumentation [12] or by executing the program in a simulator [13]. While the manual approach potentially benefits from high-level human comprehension of the source code, it is also time-consuming and error-prone. The compiler approach is automatic but the use of naive analysis techniques generates many false dependencies as its correctness relies on conservative assumptions about pointer effects. When used to allocate tasks to processing elements, the extra edges in the task graph over-constrain the problem and reduce the available mapping choices. The simulation approach relies on a single execution and does not take task dependencies over all possible program paths into account. Consequently, the result of such an approach is only valid for a particular program execution.

In comparison, our proposal complements the information that may be gathered from points-to analysis. It reduces excess edges in a task graph by capturing precise information on inter-task dependencies since all communication via pointers is either declared or causes a runtime error.

3.2 Reducing Communication between Processing Elements

Programming models supporting the abstraction of a global shared memory provide a coherent view of the memory hierarchy to all processing elements. To maintain the view of the memory hierarchy, the processing elements communicate information on memory updates among themselves. Although several optimizations exist [14,15] which reduce the communication, the overhead of communicating remains significant. Information about inter-task dependencies may be used to limit the traffic by only communicating updates to processing elements which run dependent tasks. We envision the optimizations to be useful in systems where the communication protocols maintaining the memory view can be altered to fit the requirements of applications.

3.3 Improving Program Analysis

Pointers and arrays are used in almost all programming languages. While being useful programming abstractions, they also pose problems for tools, such as compilers, that analyse programs. These tools must take the effect of operations through pointers into account. Points-to analysis [16,17] is used to approximate the effects of operations through pointers. If two pointers point to the same storage location they are said to be aliases. Similarly, the array-indexing expressions `a[i]` and `a[j]` alias if `i` equals `j`. Many algorithms performing points-to analysis exist. Each of them offers a different trade-off in terms of scalability of the algorithm and precision of the result. However, precise points-to analysis has been shown to be undecidable [18]. The limitation on precision is demonstrated by an example in Sect. 4. In a survey, Hind [17] suggests using programmer annotations to improve precision.

We believe the runtime checks which enforce the task dependency declarations, as explained in Sect. 5, can be used for this purpose. They allow a points-to analysis for an annotated program to assume that, for a task t, a pointer can only alias i) pointers which are task-private to t or ii) task-shared pointers contained in tasks with a declared dependency on t. All other aliasing will cause runtime errors.

This leads to more precise analysis. In addition, the input to the analysis algorithms becomes smaller yielding a shorter execution time. The smaller input set also makes it possible to use more advanced analysis algorithms that have a higher computational complexity.

3.4 Related Work

We are not aware of any previous work which annotates OpenMP programs to identify communication between tasks. Adve and Sakellariou [19] have implemented compiler supporting synthesis of task graphs from High-Performance Fortran programs which use MPI [20] for message passing. They also argue in favor of generating task graphs from OpenMP programs.

The Jade [8] programming language uses dependency declarations as the primary means of expressing parallelism, communication and synchronization. SmpSs [7] is a more recent proposal which aims at extending the OpenMP tasking model [21] to add support for dependent tasks using declarations with semantics similar to those in Jade.

Compared to Jade and SmpSs, the task dependency directive proposed in this paper differs by capturing not only the names of input and output *variables* but also those of input and output generating *tasks*. Furthermore, Jade and SmpSs use dependency information to dynamically schedule irregular workloads whereas our proposal uses dependencies for compile time identification of inter-task communication for the full range of parallel patterns supported by OpenMP.

4 Limitations of Points-to Analysis

Listing 1 illustrates a situation in which points-to analysis is forced to pessimistically assume that `b`, `c` and `d` may alias which is safe but also restrictive for clients of the analysis. Consequently, points-to analysis can not, in general, determine if two tasks in a shared memory program communicate or not.

One can use message passing to make communication between different program parts or tasks explicit but it is not a universal panacea however [22]. Studies comparing benchmarks using message passing, MPI [20], and shared memory programming using ccNUMA or software distributed-shared memory architectures have shown that the message passing implementations deliver the best

Listing 1. Pointer aliasing problem. Since the behavior of the function `f` can be arbitrarily complex, points-to analysis must assume that `b`, `c` and `d` may alias `a` to be safe. It is therefore unknown if the tasks `first`, `second` and `third` share data.

```
int a, *b, *c, *d, *f(void); /* f is an arbitrary function */
void first (int *x) { ... }
void second(int *y) { ... }
void third (int *z) { ... }

void g(void) {
  b = &a; c = f(); d = f();  /* b, c and d may alias */
#pragma omp parallel
  {
#pragma omp task
    first (b);
#pragma omp task
    second(c);
#pragma omp task
    third (d);
  }
}
```

Listing 2. Declaration of task dependencies using the proposed directive. The `output` clause on line 5 means that the `first` task will write data to the variable `x` which is read by tasks `second` and `third`. Similarly, task `second` declares that `y` provides input from `first` and output to a task `fourth` which is not shown.

```
 1 void g(void) {
 2   b = &a; c = f(); d = f();  /* b, c and d may alias */
 3 #pragma omp parallel
 4   {
 5 #pragma depends first  output(x,second,third)
 6 #pragma omp task
 7     first (b);
 8 #pragma depends second input(y,first) output(y,fourth)
 9 #pragma omp task
10     second(c);
11 #pragma depends third  input(z,first)
12 #pragma omp task
13     third (d);
14   }
15 }
```

performance but are also more cumbersome to write since the explicit handling of communication imposes an additional burden on the programmer [4,23,24].

4.1 Using Task Dependency Declarations

In Listing 2, the function `g` from the previous example has been adapted such that tasks mention its dependent tasks in the input and output declarations.

Compared to Listing 1, the extended declarations make it trivial to determine that the task `first` provides output to `second` and `third` and that `second` does not provide output to `third`.

As discussed in Sect. 3.4, the task dependency declarations used in Jade and SmpSs do not contain the names of dependent tasks in the input and output clauses, e.g., line 8 in Listing 2 would be reduced to `input(y) output(y)`. If task names were removed from the `input` and `output` clauses in Listing 2 it must once again be pessimistically assumed that communication is possible between all three tasks. This justifies the inclusion of task names in our proposed directive.

5 Runtime Implementation

The task dependency declarations must be enforced dynamically and this section describes how this is accomplished. Most importantly, it must be ensured that a task will not dereference a checked pointer which is not declared as a dependency. This section will also discuss the runtime calls which can be emitted by a compiler to enforce the dependency declarations.

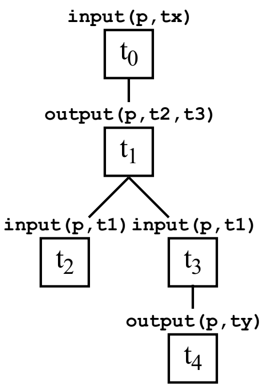
We define a function $rd(p)$ which maps a runtime checked pointer p to the set of all tasks which may read its pointee. Each task and pointer pair $\langle t, p\rangle$ is mapped to the set of tasks which are declared as readers of shared objects through p in the `depends` statement of t by $rd_{\mathrm{deps}}(p,t)$. Finally, $rd_{\mathrm{pre}}(p,t)$ is the function which gives the set of tasks which are readers of shared objects through p but may only execute before the first execution of t. Analogous definitions exists for writes through runtime checked pointers as shown in Table 1.

Before a task t reads the pointee of p it must be checked that

$$t \in rd(p) \wedge wr(p) = wr_{\mathrm{deps}}(p,t) \cup wr_{\mathrm{post}}(p,t) \tag{1}$$

Table 1. Auxiliary functions which define the sets of tasks used when describing the functioning of the runtime operations

Function	Output
$rd(p)$	the set of all tasks which may read task-shared objects pointed to by p
$wr(p)$	the set of all tasks which may write task-shared objects pointed to by p
$rd_{\mathrm{deps}}(p,t)$	the set of tasks which are declared as readers of task-shared objects pointed to by p in the `depends` statement of t
$wr_{\mathrm{deps}}(p,t)$	the set of tasks which are declared as writers of task-shared objects pointed to by p in the `depends` statement of t
$rd_{\mathrm{pre}}(p,t)$	the set of tasks which may read task-shared objects pointed to by p but never execute after or in parallel with t
$wr_{\mathrm{post}}(p,t)$	the set of tasks which may write task-shared objects pointed to by p but never execute before or in parallel with t



a

$rd(p) = \{t_0, t_2, t_3\}$ $\quad wr(p) = \{t_1, t_4\}$
$rd_{\text{pre}}(p, t_1) = \{t_0\}$ $\quad wr_{\text{post}}(p, t_3) = \{t_4\}$
$rd_{\text{deps}}(p, t_1) = \{t_2, t_3\}$ $\quad wr_{\text{deps}}(p, t_3) = \{t_1\}$

Output check of p before its pointee is written by t_1:
$t_1 \in wr(p) \wedge rd(p) = rd_{\text{deps}}(p, t_1) \cup rd_{\text{pre}}(p, t_1) \rightarrow$
$t_1 \in \{t_1, t_4\} \wedge \{t_0, t_2, t_3\} = \{t_2, t_3\} \cup \{t_0\}$

Input check of p before its pointee is read by t_3:
$t_3 \in rd(p) \wedge wr(p) = wr_{\text{deps}}(p, t_3) \cup wr_{\text{post}}(p, t_3) \rightarrow$
$t_3 \in \{t_0, t_2, t_3\} \wedge \{t_1, t_4\} = \{t_1\} \cup \{t_4\}$

b

Fig. 2. a) Program fragment showing the control flow between seven tasks that share data through a pointer p. Tasks t_x and t_y are omitted for clarity. b) Example runtime checks which apply (1) to p in t_3 and (2) to p in t_1. Although both of tasks t_1 and t_4 write to the pointee of p, tasks t_2 and t_3 need only declare an input dependency on t_1 because t_4 is always executed after t_3. Similarly, the `output` clause of t_1 need not mention t_0 as it always executes before t_1.

and similarly, it must be checked that

$$t \in wr(p) \wedge rd(p) = rd_{\text{deps}}(p, t) \cup rd_{\text{pre}}(p, t) \tag{2}$$

before the pointee of p is written to by a task t. An concrete example is shown in Fig. 2. Since the second part of the conjunction is the same for all tasks which read or write p, the necessary dependency information is stored in a global hash table in which each pair of bit-vectors $\langle rd(p), wr(p)\rangle$ is keyed on p.

A runtime check is simply a lookup on p in the dependency hash table followed by one bit-vector membership test and one bit-vector comparison corresponding to the left and right-hand side respectively of conjunction (1) or (2). Bit-vectors are represented as machine word sized elements in a fixed size array meaning that operations on bit-vectors can be supported efficiently.

To determine where runtime checks should be performed, the relaxed-consistency memory model [5] of OpenMP must be taken into account. All the *task-boundary* points defined in Sect. 2 implies a `flush` operation. Consequently, given a conforming OpenMP program, a single runtime check inserted between each `flush` and dereference of a runtime checked pointer p is sufficient to detect violations of a task dependency declaration.

Besides the runtime checks themselves, we have implemented the following operations which update the dependency hash table.

`register_input(p, input_tasks)` Sets the initial value of $rd(p)$ in the dependency table. Called at program start-up to register checked, file-scope pointers. It is also called at runtime to set dependencies of runtime checked pointers to dynamically allocated memory.

`register_output(p, output_tasks)` Sets the initial value of $wr(p)$. Otherwise it is similar to `register_input`.

`update_input(p, input_tasks)` Updates $rd(p)$ in the dependency table such that the new value equals $rd(p) \cup input\_tasks$. Called to update the input dependencies $rd'(p_1) \leftarrow rd(p_1) \cup rd(p_2)$ when a runtime checked pointer p_1 is assigned the value of a runtime checked pointer p_2.

`update_output(p, output_tasks)` Updates $wr(p)$. Otherwise it is similar to `update_input`.

`unregister(p)` Removes $rd(p)$ and $wr(p)$ in the dependency table. Called immediately after dynamic memory deallocation.

Accesses to the dependency table need to be synchronized since it is potentially read and updated concurrently.

We currently use a read/write lock [25] provided by the POSIX threads [26] implementation on our platform. This means that the runtime checks can happen concurrently by acquiring the lock for reading while the register, update and remove operations may need to update the dependency table under a write lock. The unregister operation always acquires the lock for writing. Since the time to acquire the lock for reading is several orders of magnitude slower than a lookup in the dependency table, the synchronization overhead is amortized by only acquiring the lock for reading once for all necessary runtime checks at a task-boundary.

6 Results

We have conducted two types of experiments. First, the cost and scalability of each of the runtime operations were investigated. Second, the effort to annotate a sample program and the performance impact of the runtime checks were examined.

All benchmarks were compiled with GCC 4.3.2 with the optimizations implied by the `-O2` flag and run under the Ubuntu 8.10 Linux distribution using a 2.66 GHz Intel Core i7 CPU. Due to slight variations in the results of the individual benchmark runs, we use averages calculated from thirty consecutive benchmark executions.

6.1 Runtime Operations

Each of the runtime operations discussed in Sect. 5 were benchmarked and the results are shown in Fig. 3. Each operation was executed repeatedly in a tight loop to increase execution times well above the timer resolution.

The measurements of the operations which may insert a key into the dependency table is split into two cases: key present and key not present. To compare the cost of the runtime operations with the cost of entry and exit of a task-boundary, the performance of the OpenMP `parallel` construct is also included in the graph. It was measured using the EPCC micro-benchmark suite [27].

The `check_input` and `check_output` takes at most two microseconds. Since the cost is dominated by the time to acquire the lock for reading, this value is also

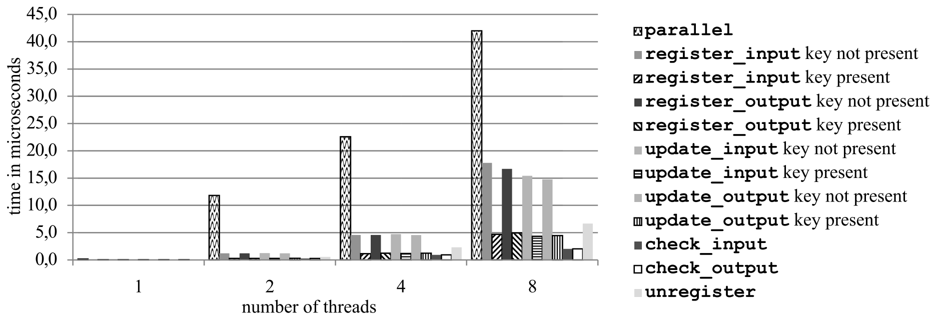


Fig. 3. Time taken by a single execution of each runtime operation and the OpenMP `parallel` directive in microseconds as a function of the number of threads

a good indicator of the total cost incurred by all checks at a task-boundary such as entry to a `parallel` section. Thus, we expect the added cost of performing runtime checks at task-boundaries to be insignificant.

Replacing the read/write lock with read-copy-update synchronization [28] would allow read operations to become lock-free. As read-only runtime checks typically execute much more frequently than the other operations we expect this to significantly lower the overhead and improve the scalability of our implementation.

The register and update operations which need not insert a key into the dependency table take at most five microseconds and reflects the cost of creating an alias of a checked pointer which is already registered with the runtime. Unregistering a shared object takes slightly longer but is only necessary when deallocation of the object which itself is typically a costly operation.

Finally, the operations which need to insert a key into the dependency table under a write-lock have the highest execution time which is approximately fifteen microseconds in case of eight threads. Again, these more costly operations occur only in context of memory allocation which itself may be costly and thus is likely to be used sparingly by a skilled programmer.

For all operations except `omp parallel`, `check_input` and `check_output`, a super-linear increase in execution time is observed when going from four to eight threads. This is most likely explained by the fact that in the latter case two threads must share the resources of a single processor core as the benchmark system only contains four processor cores even though it can execute eight threads simultaneously.

6.2 Sample Application: Integer Sort

The task dependency directives were applied to the integer sort kernel, IS, in the NAS parallel benchmarks[1] [3,4] to determine the impact on programming effort and runtime performance. The IS program performs integer sort on a large array

[1] `http://www.nas.nasa.gov/Resources/Software/npb.html` was used to obtain version 3.3 of the NAS parallel benchmarks.

and tests integer computation as well as communication performance [3]. The workload is distributed primarily by means of loop-parallelism.

First we identified the set of task boundaries. Then the set of runtime checked pointers was identified and `depends` statements were added. Finally, the calls required to register, update and check task dependencies were added by hand as we have yet to automate this step.

The source contained 4 `parallel` sections, 5 worksharing sections which partitions the program into 14 tasks. There were 8 runtime checked pointers. We inserted 7 calls to `register_input`, 8 to `register_output`, 3 to `update_input`, 1 to `update_output` and finally 19 and 11 calls to `check_input` and `check_output` respectively. It was not necessary to call `unregister` as no memory was deallocated.

The execution time of the instrumented and un-instrumented versions of the IS kernel was measured using the GNU `time` utility. The timing facility built into the benchmark was not used as it would not account for the cost of initializing our runtime. The benchmark was executed repeatedly for class *S, A* and *B* workloads which sorts 2^{16}, 2^{23} and 2^{25} 32-bit integers respectively. For none of the workloads were there a detectable change in execution time above the measurement accuracy.

7 Conclusions

We have presented an extension to OpenMP that makes it possible to declare inter-task communication patterns. The `depends` construct increases the amount of knowledge of inter-task communication which is available statically. The technique presented in this paper will pessimistically overestimate the communication between tasks in two cases. First, we do not take array slicing into account and rather assume the entire array to be shared. Secondly, we assume that shared data structures are partially updated and ignore the possible optimizations that can be done if a data structure is written in its entirely by a task.

Our technique was evaluated using the IS benchmark from the NPB benchmark suite. The measurement shows no performance degradation from the runtime checks that was inserted into the program code to test if the program behavior conform to the specified communication patterns. The result was generated using an Intel Core i7-based workstation supporting the execution of up to eight threads in parallel.

We also used a set of micro-benchmarks to further evaluate the cost and scalability of the individual runtime operations. It was found that even though the execution time is dominated by the time taken to acquire locks, the cost of the runtime checks executed at entry to a task is an order of magnitude lower than the cost of entering and exiting an `omp parallel` section.

Acknowledgements

This work has been partially supported by HiPEAC^2 and ArtistDesign, both European Union Networks of Excellence. The authors would like to thank Michael R. Hansen for providing thoughtful comments and feedback.

References

1. OpenMP Architecture Review Board: OpenMP application program interface, version 3.0. Technical report, OpenMP Archtecture Review Board (2008)
2. Liblit, B., Aiken, A., Yelick, K.A.: Type systems for distributed data sharing. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 273–294. Springer, Heidelberg (2003)
3. Bailey, D.H., et al.: The NAS parallel benchmarks—summary and preliminary results. In: Proceedings of Supercomputing 1991, pp. 158–165. ACM, New York (1991)
4. Jin, H., Frumkin, M., Yan, H.: NPB-OpenMP 3.0. Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA 94035-1000 (October 1999)
5. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
6. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley-Interscience, Hoboken (2007)
7. Duran, A., Pérez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP tasking model to allow dependent tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
8. Rinard, M.C., Lam, M.S.: The design, implementation, and evaluation of Jade. TOPLAS 20(3), 483–545 (1998)
9. Allen, F.E.: Control flow analysis. SIGPLAN Not. 5(7), 1–19 (1970)
10. Schmitz, M.T., Al-Hashimi, B.M., Eles, P.: System-Level Design Techniques for Energy-Efficient Embedded Systems. Kluwer Academic Publishers, Norwell (2004)
11. Sinnen, O., Sousa, L.: A classification of graph theoretic models for parallel computing. Technical Report RT/005/99, Instituto Superior Tecnico, Technical University of Lisbon (1999)
12. Vallerio, K.S., Jha, N.K.: Task graph extraction for embedded system synthesis. In: Proceedings of VLSID 2003, p. 480. IEEE Computer Society Press, Los Alamitos (2003)
13. Liu, A.H., Dick, R.P.: Automatic run-time extraction of communication graphs from multithreaded applications. In: Proceedings of CODES+ISSS 2006, pp. 46–51. ACM, New York (2006)
14. Dubois, M., Scheurich, C., Briggs, F.: Memory access buffering in multiprocessors. SIGARCH Comput. Archit. News 14(2), 434–442 (1986)
15. Gharachorloo, K., et al.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of ISCA 1990, pp. 15–26 (1990)
16. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI 1994, pp. 242–256 (1994)
17. Hind, M.: Analysis: Havent we solved this problem yet. In: Proceedings of PASTE 2001, pp. 54–61. ACM, New York (2001)
18. Ramalingam, G.: The undecidability of aliasing. TOPLAS 16(5), 1467–1471 (1994)
19. Adve, V.S., Sakellariou, R.: Compiler synthesis of task graphs for parallel program performance prediction. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 208–226. Springer, Heidelberg (2001)
20. Snir, M., et al.: MPI – The Complete Reference, 2nd edn. The MPI Core, vol. 1. The MIT Press, Cambridge (1998)

21. Ayguadé, E., et al.: A proposal for task parallelism in OpenMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 1–12. Springer, Heidelberg (2008)
22. Asanovic, K., et al.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
23. Rodman, A., Brorsson, M.: Programming effort vs. performance with a hybrid programming model for distributed memory parallel architectures. In: Proceedings of EuroPar 1998, pp. 888–898 (1999)
24. Karlsson, S., Brorsson, M.: A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2. In: Panda, D.K., Stunkel, C.B. (eds.) CANPC 1998. LNCS, vol. 1362, pp. 189–201. Springer, Heidelberg (1998)
25. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
26. IEEE: IEEE std. 1003.1c-1995 thread extensions. Technical report, IEEE, Formerly POSIX.4a. now included in 1003.1-2004 (1995)
27. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. SIGARCH Comput. Archit. News 29(5), 41–48 (2001)
28. McKenney, P.E., Slingwine, J.D.: Read-copy update: Using execution history to solve concurrency problems. In: Proceedings of PDCS 1998, pp. 509–518 (1998)

Author Index