Mitsuhisa Sato Toshihiro Hanawa
Matthias S. Müller Barbara M. Chapman
Bronis R. de Supinski (Eds.)

Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More

LNCS 6132

6th International Workshop on OpenMP, IWOMP 2010
Tsukuba, Japan, June 2010
Proceedings

Springer

Lecture Notes in Computer Science 6132

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Mitsuhisa Sato Toshihiro Hanawa
Matthias S. Müller Barbara M. Chapman
Bronis R. de Supinski (Eds.)

Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More

6th International Workshop on OpenMP, IWOMP 2010
Tsukuba, Japan, June 14-16, 2010
Proceedings

Springer

Volume Editors

Mitsuhisa Sato
Toshihiro Hanawa
E-mail: msato@cs.tsukuba.ac.jp, hanawa@ccs.tsukuba.ac.jp

Matthias S. Müller
E-mail: matthias.mueller@tu-dresden.de

Barbara M. Chapman
E-mail: chapman@cs.uh.edu

Bronis R. de Supinski
E-mail: bronis@llnl.gov

Library of Congress Control Number: 2010927500

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-13216-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-13216-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This book contains the proceedings of the 6th International Workshop on OpenMP held in Tsukuba City, Japan, in June 2010. The International Workshop on OpenMP is an annual series of workshops dedicated to the promotion and advancement of all aspects focusing on parallel programming with OpenMP.

OpenMP is now a major programming model for shared memory systems from multi-core machines to large-scale servers. Recently, new ideas and challenges have been proposed to extend OpenMP framework to support accelerators and also to exploit other forms of parallelism beyond loop-level parallelism.

The workshop serves as a forum to present the latest research ideas and results related to this shared memory programming model. It also offers the opportunity to interact with OpenMP users, developers and the people working on the next release of the specification.

In response to the Call-for-Papers for the technical program, the Program Committee received a total of 23 submissions from all over the world including Asia, USA and Europe, and all submissions were carefully refereed in a rigorous process which required at least three reviews for each paper, using the EasyChair conference system. The final decisions were collectively made in March 2010. Due to time and space limitations for the workshop and proceedings, only 13 papers could be selected for presentation and inclusion in the proceedings. We believe we have chosen a diverse, high-quality set of papers, reflecting a stimulating and enjoyable workshop.

Finally, we would like to thank all authors, referees, and committee members for their outstanding contributions which have ensured a continuation of the high quality of IWOMP workshops.

June 2010

Mitsuhisa Sato
Toshihiro Hanawa
Matthias Müller
Barbara Chapman
Bronis R. de Supinski

Organization

IWOMP 2010 Committee

Program and Organizing Chair

Mitsuhisa Sato	University of Tsukuba, Japan

Poster and Vice Organizing Chair

Toshihiro Hanawa	University of Tsukuba, Japan

Sponsors Contact Chair

Barbara Chapman	University of Houston, USA

Tutorials Chair

Ruud van der Pas	Sun Microsystems

Organizing Committee

Mitsuhisa Sato	University of Tsukuba, Japan
Toshihiro Hanawa	University of Tsukuba, Japan
Taisuke Boku	University of Tsukuba, Japan
Daisuke Takahashi	University of Tsukuba, Japan

Program Committee

Mitsuhisa Sato	University of Tsukuba, Japan
Matthias Müller	ZIH, TU Dresden, Germany
Dieter an Mey	RWTH Aachen University, Germany
Eduard Ayguadé	Barcelona Supercomputing Center (BSC), Spain
Mark Bull	EPCC, UK
Barbara Chapman	University of Houston, USA
Bronis R. de Supinski	NNSA ASC, LLNL, USA
Guang R. Gao	University of Delaware, USA
Rick Kufrin	NCSA/University of Illinois, USA
Federico Massaioli	CASPUR, Italy
Larry Meadows	Intel, OpenMP CEO
Ruud van der Pas	Sun Microsystems
Michael Wong	IBM
Alistair Rendell	ANU, Australia
Wenguang Chen	Tsinghua University, China

Sik Lee	KISTI, Korea
Hidetoshi Iwashita	Fujitsu, Japan
Raymond Namyst	University of Bordeaux 1, France
Toshihiro Hanawa	University of Tsukuba, Japan

Steering Committee Chair

Matthias S. Müller	University of Dresden, ZIH, Germany

Steering Committee

Bronis R. de Supinski	NNSA ASC, LLNL, USA
Dieter an Mey	CCC, RWTH Aachen University, Germany
Eduard Ayguadé	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
Lawrence Meadows	KSL Intel, USA
Arnaud Renard	University of Reims, France
Mitsuhisa Sato	University of Tsukuba, Japan
Sanjiv Shah	Intel, USA
Ruud van der Pas	Sun Microsystems, Geneva, Switzerland
Matthijs van Waveren	Fujitsu, France
Michael Wong	IBM, Canada
Weimin Zheng	Tsinghua University, China

Table of Contents

Sixth International Workshop on OpenMP IWOMP 2010

Runtime and Optimization

Proposed Extensions to OpenMP

Scheduling and Performance

Hybrid Programming and Accelerators with OpenMP

Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC

Patrick Carribault, Marc Pérache, and Hervé Jourdren

CEA, DAM, DIF, F-91297 Arpajon, France
{patrick.carribault,marc.perache,herve.jourdren}@cea.fr

Abstract. With the advent of multicore- and manycore-based supercomputers, parallel programming models like MPI and OpenMP become more widely used to express various levels of parallelism in applications. But even though combining multiple models is possible, the resulting performance may not reach expected results. This is mainly due to collaboration issues between the runtime implementations. In this paper, we introduce an extended taxonomy of hybrid MPI/OpenMP programming and a new module to the MPC framework handling a fully 2.5-compliant OpenMP runtime completely integrated to an MPI 1.3 implementation. The design and implementation guidelines enable two features: (i) built-in oversubscribing capabilities with performance comparable to state-of-the-art implementations on pure OpenMP benchmarks and programs, and (ii) the possibility to run hybrid MPI/OpenMP applications with a limited overhead due to the mix of two different programming models.

1 Introduction

The advent of multicore processors as basic-blocks of cc-NUMA (*cache-coherent Non-Uniform Memory Access*) architectures leads to the necessity to extract a large amount of parallelism. Parallel programming models have emerged to help programmers expressing several kinds of parallelism (task, data, etc.) at multiple granularity levels. One of the most commonly-used model is MPI [1], representing every task as process and using message-passing methods to communicate through explicit function calls (distributed memory paradigm). Another parallel programming model is OpenMP [2]: it enables multithreaded parallelization of any part of the program by adding directives to the sequential source code.

Hybrid programming with MPI and OpenMP (or *mixed-mode programming*) is a promising solution taking advantage of both models: for example, exploiting intra-node shared-memory with OpenMP and inter-node network with MPI. The MPI model allows to exploit a large number of cluster nodes. Furthermore, the intra-node communications are usually optimized thanks to shared memory. On a NUMA architecture, this programming model implicitly creates a well-balanced data locality because every task allocates its own set of data inside its private memory. On the other hand, OpenMP fully benefits from the shared memory available inside a node by avoiding any kind of communications: data are shared by default and the programmer has to guarantee the determinism of

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 1–14, 2010.

data races (with locks or any sort of synchronization mechanisms). This leads to a gain in memory and sometimes in performance but the main drawback is the data locality on a NUMA architecture: the OpenMP standard gives no rules about parallel memory allocation. Moreover, multithreading with OpenMP enables a lightweight load-balancing mechanism within a node. But even if these two models are relying on different paradigms, mixing them does not always lead to a performance gain. Side effects appear not only because of application semantics but because of the separate implementations resulting in some extra overhead. Several studies try to analyze their corresponding behavior but none of them proposed a unified representation of both models.

This paper presents an extension to the MPC framework [3] to create a unified representation of hybrid MPI/OpenMP parallelism. MPC is a thread-based framework unifying parallel programming models for HPC. It already provides its own thread-based MPI implementation [4]. To enable hybrid parallelism, we added an OpenMP runtime and optimized it to lower the overhead due to programming-model mixing. This paper makes the following contributions: (i) the implementation of a full OpenMP runtime with oversubscribing capabilities, (ii) an extended taxonomy about approaches to adopt hybrid parallelism and (iii) a unified runtime for low-overhead hybrid MPI/OpenMP parallelism.

This paper is organized as follows: Section 2 describes the related work for these parallel programming models. Section 3 exposes an extended taxonomy of the hybrid MPI/OpenMP programming model. It details the different ways to combine these two models and their issues. Section 4 depicts the implementation of hybrid parallelism in MPC, including the design and implementation of the OpenMP runtime. Finally, Section 5 proposes some experimental results on both OpenMP and hybrid benchmarks, before concluding in Section 6.

2 Related Work

Hybridization of parallel programming models has already been explored and tested to exploit current architectures. Some applications benefit from combining MPI and OpenMP programming models [5] while some work concludes that the *MPI everywhere* or *Pure MPI* mode is the best solution [6]. This is because several issues appear when dealing with these programming models. For example, the mandatory thread-safety of the underlying MPI implementation might lead to an additional overhead [7,8]. Furthermore, the way the application combines MPI and OpenMP directly influence the resulting performance. For example, the inter-node network bandwidth may not be fully used if only a small number of threads or tasks are dealing with message-passing communications [9]. To avoid this limitation, Viet *et al.* studied the use of thread-to-thread communications [10]. The OpenMP model can be as well extended to increase the support of other programming models. Jin *et al.* explored the notion of *subteams* with different ways of using OpenMP to increase hybrid performance [11].

To the best of our knowledge, one of the most advanced study on hybridization has been proposed by Hager *et al.* [12]. They designed a taxonomy on the ways

to combine MPI and OpenMP inside the same scientific application. But all these approaches and tests only deal with simple hybrid parallelism. Indeed, they hardly explored the potential benefit of using MPI and OpenMP with oversubscribing.

3 Hybrid Programming MPI/OpenMP

This section describes the multiple ways to mix these two programming models by extending the taxonomy introduced in [12] and details some issues related to the implementation of such models.

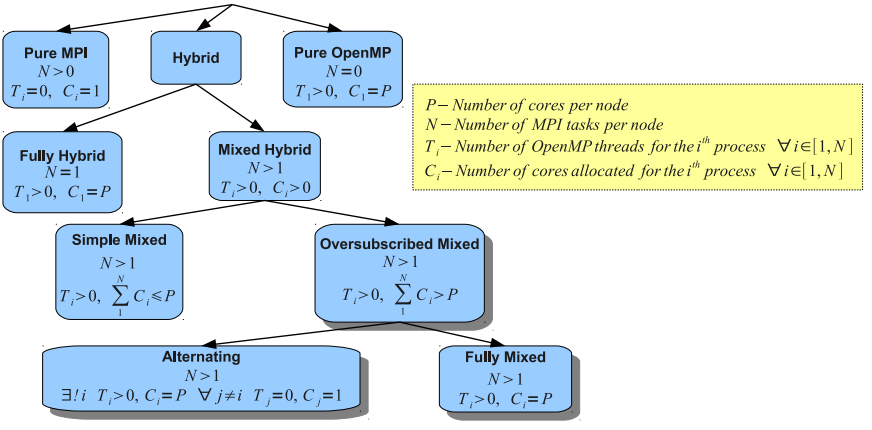


Fig. 1. Extended Taxonomy of Hybrid MPI/OpenMP Programming Models

3.1 Taxonomy of Hybrid Applications

Figure 1 proposes an extended taxonomy of hybrid parallelism [12]. We consider nodes with P cores. The classification is related to the number of MPI tasks N located on each node, the number of OpenMP threads T_i for the i^{th} process and the number of cores allocated for the i^{th} process C_i i.e., the cores on which the OpenMP threads will be spawned during the parallel regions of the i^{th} process.

Starting from the root, the first level of this tree illustrates the basic approaches: the *Pure MPI* mode uses the MPI programming model ($N > 0$) and creates no OpenMP thread at all ($T_i = 0$). On the other hand, the *Pure OpenMP* mode only spawns OpenMP threads to parallelize the application. This mode can be extended to multiple nodes [13]. The *Hybrid* subtree represents the combination of MPI and OpenMP inside the same application. The *Fully Hybrid* approach exploits the node with OpenMP threads ($T_i > 0$) and uses message-passing functions to communicate over the whole cluster ($N = 1$). The overall performance mainly relies on OpenMP parallelization on each node and might therefore be limited by Amdahl's law both because of the remaining sequential

parts (related to the fork-join model of OpenMP) and the serialized MPI communications. Deploying more than one MPI task on the node is called *Mixed Hybrid*. In this mode, it is possible to either exploit each core with one execution flow (*Simple Mixed* where the total number of allocated cores is at most P) or oversubscribe the cores to recover latencies (*Oversubscribed Mixed* subtree). The first solution is commonly used when dealing with hybrid parallelism. For example, each multicore processor can be exploited with one MPI task spawning OpenMP threads on it. The oversubscribed approaches are less explored: one advantage is the ability to recover different kinds of latencies (load-balancing, MPI communications, system calls, etc.) within the application.

This paper introduces the *Oversubscribed Mixed* subtree with two approaches: the *Alternating* mode where the application alternatively uses MPI and OpenMP parallelization, and the *Fully Mixed* mode where one MPI task is placed by core and all tasks might exploit the entire node when entering a parallel region. These approaches advocate for a solid and coherent implementation of MPI and OpenMP handling oversubscription.

3.2 Programming Model Interleaving

The taxonomy depicted in Figure 1 sorts several functional approaches to mix OpenMP threads and MPI tasks. But even if these models are not relying on the same paradigms, combining them in practice can be very tricky. First of all, the MPI library has to handle multiple concurrent control flows. For this purpose the MPI 2.0 standard introduces several levels of thread support: to fully use any of the approaches described in Figure 1, the underlying library has to support the `MPI_THREAD_MULTIPLE` level. Moreover, the task/thread placement is a real issue. Deciding where to pin such MPI task or such OpenMP thread mainly depends on the use of these models. If the application spends almost 100% of its consumed time in OpenMP parallel regions, the right solution is probably to span the OpenMP threads on each socket and use, for example, one MPI task per multicore processor (*Simple Mixed* mode). But this is not always the best approach to use hybrid programming. If one wants to use OpenMP threads to balance the load among the MPI tasks, spanning the threads everywhere and overlapping the OpenMP instances might be one approach to reach higher performance (*Oversubscribed Mixed* subtree in Figure 1). Because of this last point, the MPI and OpenMP runtimes must cooperate to establish some placement and oversubscribing policies. Indeed, load balancing can be obtained only if the corresponding runtimes allow the scheduling of one task and several threads on each core without a too large overhead. For example, if both runtime implementations use *busy waiting* to increase the reactivity and the performance of one programming model, the combination of several models might lead to an extremely large overhead, disabling the potential performance gain. All these issues are highlighted and evaluated on MPI/OpenMP implementations in Section 5.

4 MPC-OpenMP: MPC's OpenMP Runtime

To enable hybrid parallelism in MPC [3,4], we add a complete OpenMP runtime through a new module called MPC-OpenMP supporting the 2.5 standard.[1] This section details the design and implementation of the MPC-OpenMP module and its integration with the other programming models supported in MPC (Pthread, MPI).

4.1 Runtime Design and Implementation

MPC already provides a thread-based implementation of the MPI 1.3 standard [4]: it represents every MPI task/process with user-level threads (called *MPC tasks*). Thanks to *process virtualization*, every MPI task is a thread instead of being a UNIX process. The main guideline to support the OpenMP standard is to model the threads like MPI tasks. In such way, the main scheduler would see every thread and task the same way. Another constraint is the weight of OpenMP threads: it is possible to often enter and exit parallel regions with the OpenMP model. Therefore, each thread has to be very light and to be ready within a short period of time. For this purpose, we introduce in MPC a new kind of thread called *microVP* (micro Virtual Processor) scheduling its own *microthreads*. Microthreads can be seen as some sort of filaments [14] or lazy threads [15]: entering the parallel region, OpenMP threads are distributed as microthreads among microVPs. At the very beginning, microthreads do not have their own stack, they use the microVP's one when they are scheduled. This accelerates the context switches. Stacks are created on-the-fly if one thread encounters a scheduling point (e.g., any synchronization point like a barrier).

Figure 2 depicts an example of the MPC execution model when using the OpenMP programming model in the *Pure OpenMP* mode. In this example, the underlying architecture has 4 cores modeled in MPC through Virtual Processors. The main sequential control flow is managed by the MPC task on the left of the figure. This task is scheduled on the first core. When starting the application, the MPC-OpenMP module creates microVPs for each task. In this example, because there is one control flow (one task on the first core), 4 microVPs are created, one on each core. When entering an OpenMP parallel region, the MPC task distributes microthreads on the microVPs. The figure illustrates the execution model after entering a parallel region with 6 OpenMP threads. The first and second cores have to schedule 2 microthreads while the other ones only have 1 microthread to execute.

The MPC-OpenMP runtime contains optimizations related to oversubscription (i.e., more than on microthread per microVP). For example, every barrier among microthreads on the same microVP can be resolved without any synchronization mechanism.

[1] MPC 2.0 including the MPC-OpenMP module is available at `http://mpc.sourceforge.net`

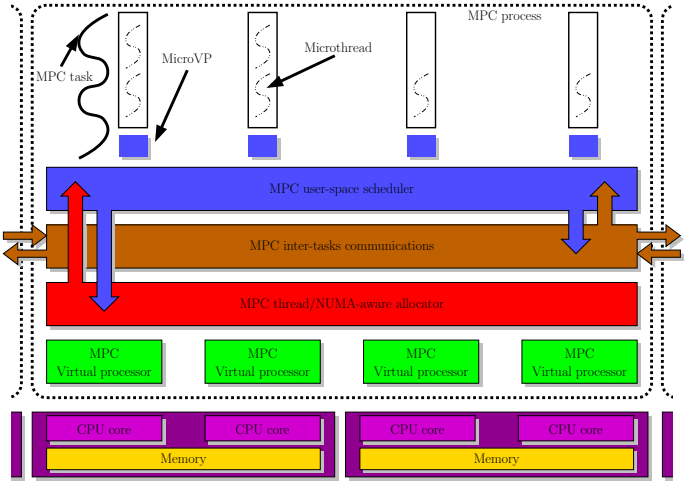


Fig. 2. MPC Execution Model of *Pure OpenMP* mode (6 OpenMP Threads on 4 Cores) adapted from [3]

4.2 Compiler Part

A compliant OpenMP implementation relies on a compiler to lower the source-code directives (e.g., `#pragma`). We modified the GCC compiler chain: instead of generating code for the GCC OpenMP library (called GOMP), the OpenMP directives are transformed into MPC internal calls (functions + Thread Local Storage). This patched GCC, called *MPC-GCC*, is available for versions 4.3.0 and 4.4.0. Thanks to these modifications, MPC supports the OpenMP 2.5 standard for C, C++ and Fortran languages.

4.3 Hybrid Execution Model

Hybrid programming is a combination of MPC tasks and threads. Each MPI task is modeled through an MPC task (user-level thread) and each instance of OpenMP is controlled by a set of microVPs scheduling microthreads when entering a parallel region. Each MPI task has its own set of microVPs to span the OpenMP threads on the node. The way these microVPs are initially created directly influences the mixing depth of the programming models. If every MPI task creates microVPs on the cores located on the same CPU socket, every OpenMP parallel region will span on the same multicore processor (*Simple Mixed* mode). This feature allows to limit the oversubscribing factor and maximizes locality. Figure 2 can be seen as an example of *Fully Hybrid* approach if the MPC task is actually an MPI task.

To enable hybrid parallelism with low overhead, some optimizations have been performed on the OpenMP and MPI parts of MPC. First of all, MPC has been

extended with `MPI_THREAD_MULTIPLE` features. The MPI standard requires to respect the message posting order. Nevertheless, once the matching phase is done, the message copies from the send buffer into the receive buffer can be performed concurrently. MPC-MPI has been optimized to parallelize the copy phase allowing multiple message copies in parallel of matching. Communications involving small messages were optimized using an additional copy into a temporary buffer to immediately release the sender. Initially, there was a unique buffer for each MPI task. This buffer has been duplicated for each microVP and the lock has been removed. MPC-OpenMP has also been optimized to deal with MPC-MPI. Busy waiting (e.g., when spawning parallel region) has been replaced by a polling method integrated in the MPC user-space scheduler leading to more reactivity without disturbing the other tasks.

5 Experimental Results

This sections describes the experiments to evaluate and validate the hybrid capabilities of MPC according to the taxonomy Figure 1. We first present the performance of the OpenMP runtime and the MPC-GCC compiler. Then, we detail the results of combining the MPI and OpenMP models on several representative benchmarks.

5.1 Experimental Environment

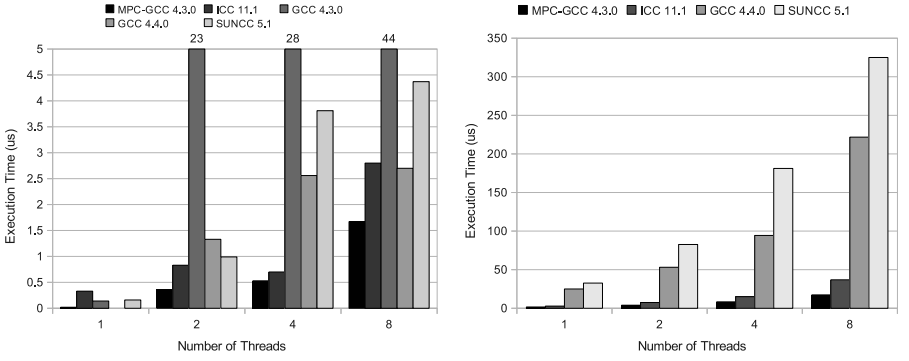
The experimental results detailed in this paper were obtained on a dual-socket quad-core Nehalem EP node[2] (8 cores) with 24GB of memory running under a Linux operating system. For lack of space, we only present the results on this architecture. We conducted the same experiments on Core2Duo and Core2Quad architectures and observed similar behaviors. For every benchmark, we bind the MPI tasks through the `schedaffinity` interface or other options (e.g., environment variables for IntelMPI). For example, IntelMPI provides several environment variables to place the tasks on the node and to reserve some cores for every task according to the topology of the underlying architecture.

5.2 Performance of MPC-OpenMP and MPC-GCC

This section describes the results of the OpenMP runtime implemented in MPC in *Pure OpenMP* mode. The compiler part is done through MPC-GCC version 4.3.0 and 4.4.0. Our implementation is measured against state-of-the-art compilers with their associated OpenMP runtime: Intel ICC version 11.1, GCC version 4.3.0 and 4.4.0, and Sun Compiler version 5.1.

EPCC Micro-Benchmarks: EPCC is a benchmark suite to measure the overhead of every OpenMP construct [16]. For each implementation, we tried the best

[2] Hyper-Threading and Turbo Boost are disabled.



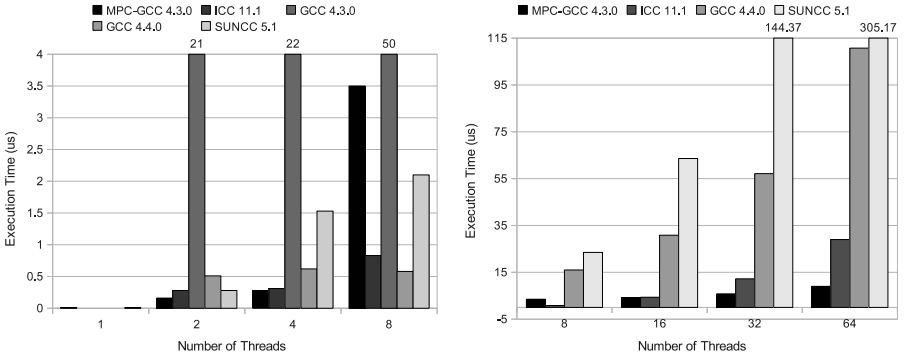
-a- From 1 to 8 threads

-b- From 8 to 64 threads

Fig. 3. Overhead of OpenMP Parallel Construct on 8-core Nehalem Architecture

option combination (CPU affinity, waiting mode, ...). Figure 3 shows the overhead of opening and closing a parallel region (`#pragma omp parallel`) according to the total number of OpenMP threads. Figure 3-a depicts the overhead from 1 to 8 threads. Performance of MPC-GCC 4.3.0 and MPC-GCC 4.4.0 are similar: for this benchmark, only the OpenMP runtime shapes the performance, not the compiler. The performance of MPC are slightly better than the OpenMP implementation of ICC 11.1, GCC 4.4.0 and SunCC 5.1 though. GCC 4.3.0 gets some very surprising results: the overhead of almost every OpenMP construct is far behind the other implementations we tested. Thus, it does not appear on Figure 3-b. This figure describes the oversubscribing performance of the OpenMP runtimes. We tested up to an oversubscribing factor of 8 (i.e., 64 threads on 8 cores). The overhead of the parallel construct in the GCC 4.4.0 and SunCC implementations become very large compare to MPC, ICC. MPC implementation stays linear while ICC OpenMP results in an exponential overhead. Note these figures depict results with the best option combination. For example, we use the `SUNW_MP_THR_IDLE` option to specify the waiting mode of SunCC and the *passive mode* (`OMP_WAIT_POLICY` environment variable) for GCC 4.4.0 for the oversubscribing mode. This is the most suitable behavior for hybrid parallelism but this adaptation has a huge impact on the raw OpenMP performance: the overhead of entering and exiting parallel regions is 10 times larger in the passive mode without oversubscribing.

Figure 4 depicts the time consumed (in micro-seconds) by an explicit barrier (`#pragma omp barrier`) according to the number of threads. The same remarks can be made on the performance of GCC 4.3.0. The MPC-OpenMP runtime has a larger overhead on 8 cores than ICC, GCC 4.4 and SunCC. This is probably due to the NUMA aspect of the architecture. Even though the MPC runtime has NUMA-aware features, the barrier implementation does not fully exploit them. It might be interesting to implement a more efficient algorithm [17]. Nevertheless, the MPC-OpenMP oversubscribing performance depicted in Figure 4-b shows strong improvements compared to ICC, GCC and SunCC.



-a- From 1 to 8 threads -b- From 8 to 64 threads

Fig. 4. Overhead of OpenMP Barrier Construct on 8-core Nehalem Architecture

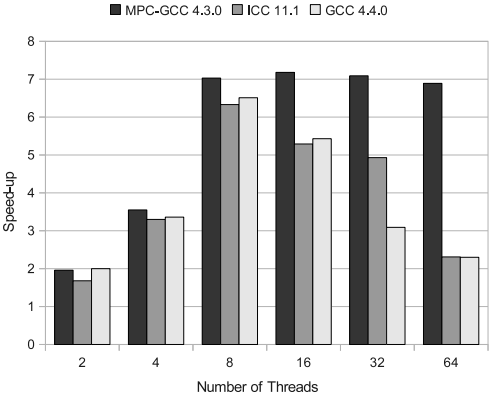


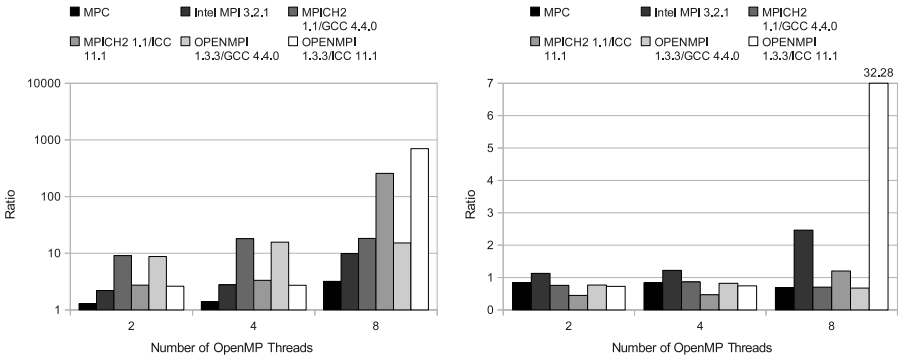
Fig. 5. BT Benchmark Results on 8-core Nehalem Architecture

NAS Benchmarks: Figure 5 shows the speed-ups obtained for the BT benchmark of the NAS 3.3 OpenMP suite [18]. Each acceleration is related to the corresponding single-threaded OpenMP version. MPC-GCC 4.4.0 obtains the same performance as MPC-GCC 4.3.0. The oversubscribing performance of MPC (with a number of threads larger than 8) are stable compared to other OpenMP implementations (ICC and GCC). Running this benchmark with 16 threads (i.e., 2 OpenMP threads per core) leads to slightly better performance than 8 with MPC. Indeed, the overall best speed-up is 7 reached by MPC with an oversubscribing factor of 2.

5.3 Overhead of Hybrid Parallelism

To evaluate the overhead of hybrid MPI/OpenMP parallelism, we designed a set of benchmarks related to the taxonomy depicted in Figure 1. Parts of these

benchmarks are directly extracted from [19]. We compare combinations of MPI libraries and OpenMP runtimes/compilers: IntelMPI, MPICH2 1.1 and OPENMPI 1.3.3 with ICC 11.1 and GCC 4.4.0. The target architecture is the one defined in Section 5.1.



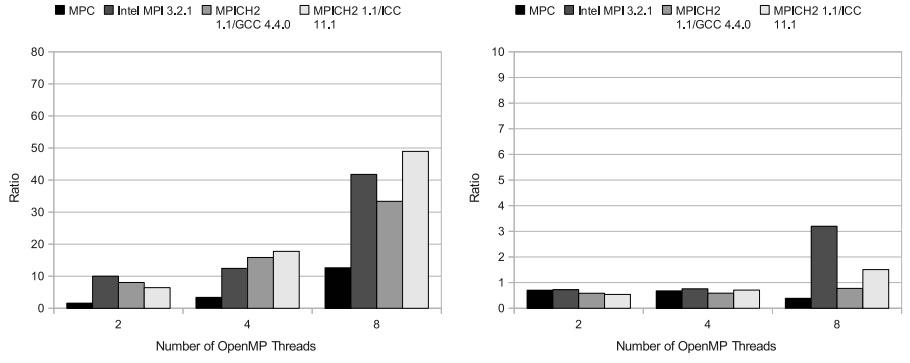
-a- 1KB messages -b- 1MB messages

Fig. 6. Overhead of Funneled Hybrid Ping-Pong on Nehalem Architecture

Funneled Hybrid Ping-Pong: Figures 6-a and 6-b depict the overhead of hybrid MPI/OpenMP parallelism for an hybrid Ping-Pong benchmark [19] with MPI communications made by the master thread of the OpenMP region. The MPI library is required to be initialized with the `MPI_THREAD_FUNNELED` level. These plots illustrate the ratio of running a funneled ping-pong according to the performance of the single threaded version. OpenMP regions are used to read and write a buffer while the MPI communications are done by the master thread after an OpenMP barrier. Figure 6-a depicts the ratio (overhead) for messages of 1KB length. MPC has a low overhead compared to other MPI/OpenMP combinations: with 2 or 4 threads (*Simple Mixed* mode) the performance remains stable while it only increases a little with 8 threads (*Fully Mixed* mode).

Figure 6-b illustrates the same results for 1MB messages. Several implementations are able to reach better performance thanks to hybrid programming (ratio below 1). Runs with GCC OpenMP obtain decent speed-up but this is partly because it uses a passive waiting mode (for OpenMP) leading to low performance for OpenMP worksharing constructs. Nevertheless, MPC is the only hybrid implementation to reach speed-ups for all number of threads.

Multiple Hybrid Ping-Pong: The second benchmark is an hybrid ping-pong [19] with communications done by every thread inside the OpenMP parallel region. The MPI library is required to be initialized with the `MPI_THREAD_MULTIPLE` level. Figure 7-a depicts the ratio for 1KB messages compared to the single-threaded version. Note OPENMPI is not represented on this plot because we had some issues with its thread safety. The overhead of MPC is low compared



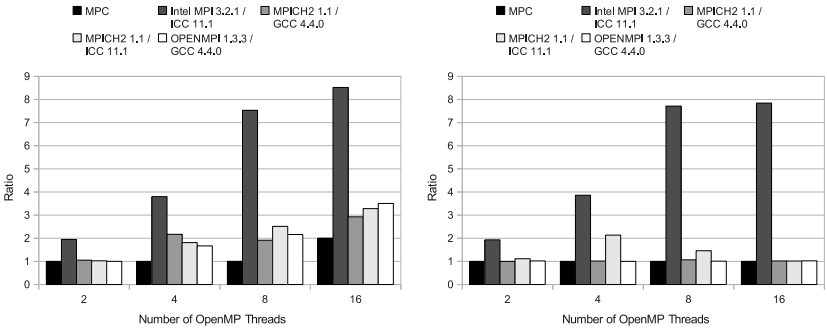
-a- 1KB messages -b- 1MB messages

Fig. 7. Overhead of Multiple Hybrid Ping-Pong on Nehalem Architecture

to combinations including MPICH2 and IntelMPI. Indeed, it does not exceed 12 with 8 threads per MPI task while it reaches up to 50 for some implementations. Figure 7-b shows the overhead for 1MB messages. Most of the combinations benefit from hybrid parallelism to accelerate the ping-pong benchmark (ratio < 1). But MPC reaches the best performance with 8 threads per MPI task while MPICH2 and OPENMPI degrade their performance (*Oversubscribed Mixed* mode).

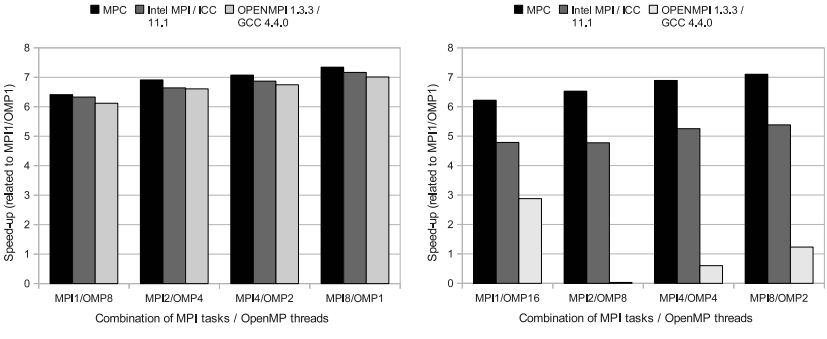
Alternating Benchmark: The last benchmark illustrates the *Alternating* mode depicted in the taxonomy tree Figure 1. Figure 8-a pictures the overhead ratio of one MPI task entering a parallel region (the threads performing independent computations) while the other 7 tasks are waiting on a barrier. While the MPC overhead is almost null up to 8 threads, other MPI/OpenMP combinations involve an overhead. Especially IntelMPI with the best performance options may degrade the execution time by a factor of 8, just because some MPI tasks are waiting on a barrier during the computational part of the single OpenMP parallel region. On the other hand, Figure 8-b shows the overhead of OpenMP computations done by the master thread of every MPI task at the same time. The overall ratio is smaller than for the previous *Alternating* benchmark except for the IntelMPI library (with ICC OpenMP) which reaches a slowdown of 8.

NAS-MZ Benchmark: Figure 9 depicts the results of running the hybrid version BT: BT-MZ version 3.2 with class B. These figures represent a total of 8 threads and a total of 16 threads (number of MPI tasks times number of OpenMP threads per task). Even though some versions reach speed-ups, the best acceleration is done thanks to 8 MPI tasks with one (*Pure MPI* mode) or 2 threads per task (*Oversubscribed Mixed* mode). Performance of MPC combining MPI and OpenMP is comparable to state-of-the-art implementations even on non-intensive hybrid benchmarks.



-a- MPI Tasks Waiting -b- OpenMP Threads Waiting

Fig. 8. Overhead of *Alternating* Benchmark on 8-core Nehalem Architecture



-a- Total of 8 threads -b- Total of 16 threads

Fig. 9. Speed-up of BT-MZ on Nehalem Architecture

6 Conclusion and Future Work

This article introduces an extension to the MPC framework to enable a unified representation of hybrid MPI/OpenMP parallelism. The new OpenMP implementation shows performance comparable to state-of-the-art implementations on current architecture (Nehalem) with efficient support of oversubscribing. This feature might be one key for future-generation hybrid applications. Moreover, we propose an extended taxonomy to explore new ways to hybridize applications and we test them on different benchmarks. The results show that MPC enables a low-overhead combination of MPI and OpenMP. MPC 2.0 including these hybrid capabilities is available at `http://mpc.sourceforge.net`

For future work, several optimizations can be done on the OpenMP runtime. For example, the barrier algorithm might be improved [17]. Other approaches can be tested inside the MPC framework like dynamic load-balancing of MPI applications with OpenMP [20] or subteam utilization [11]. OpenMP 3.0 is

becoming a new standard introducing the notion of OpenMP task and the ability to easily balance an application. Supporting OpenMP 3.0 is a future direction. Finally, hybrid performances have to be evaluated on large-scale applications.

References

1. MPI Forum: MPI: A message passing interface standard (March 1994)
2. OpenMP Architectural Board: OpenMP API (2.5 and 3.0) (May 2008)
3. Pérache, M., Jourdren, H., Namyst, R.: MPC: A unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)
4. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: An MPI implementation reducing the overall memory consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/PVI 2009. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009)
5. Chen, L., Fujishiro, I.: Optimization strategies using hybrid MPI+OpenMP parallelization for large-scale data visualization on earth simulator. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 112–124. Springer, Heidelberg (2008)
6. Lusk, E.L., Chan, A.: Early experiments with the OpenMP/MPI hybrid programming model. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 36–47. Springer, Heidelberg (2008)
7. Thakur, R., Gropp, W.: Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 46–55. Springer, Heidelberg (2007)
8. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
9. Rabenseifner, R.: Hybrid parallel programming: Performance problems and chances. In: Proceedings of the 45th CUG (Cray User Group) Conference (2003)
10. Viet, T.Q., Yoshinaga, T., Sowa, M.: Optimization for hybrid MPI-OpenMP programs with thread-to-thread communication. Institute of Electronics, Information and Communication Engineers (IEICE) Technical Eeport, 19–24 (2004)
11. Jin, H., Chapman, B., Huang, L., an Mey, D., Reichstein, T.: Performance evaluation of a multi-zone application in different openmp approaches. Int. J. Parallel Program. 36(3), 312–325 (2008)
12. Hager, G., Jost, G., Rabenseifner, R.: Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of Cray User Group (2009)
13. Hoeflinger, J.: Extending openmp to clusters (2006)
14. Engler, D.R., Andrews, G.R., Lowenthal, D.K.: Filaments: Efficient support for fine-grain parallelism. Technical report (1994)
15. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. 37(1), 5–20 (1996)
16. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. SIGARCH Comput. Archit. News 29(5), 41–48 (2001)
17. Nanjegowda, R., Hernandez, O., Chapman, B.M., Jin, H.: Scalability evaluation of barrier algorithms for OpenMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 42–52. Springer, Heidelberg (2009)

18. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report: NAS-99-011 (1999)
19. Bull, J.M., Enright, J.P., Ameer, N.: A microbenchmark suite for mixed-mode OpenMP/MPI. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 118–131. Springer, Heidelberg (2009)
20. Corbalán, J., Duran, A., Labarta, J.: Dynamic load balancing of MPI+OpenMP applications. In: ICPP, pp. 195–202. IEEE Computer Society, Los Alamitos (2004)

A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries*

Chunhua Liao, Daniel J. Quinlan, Thomas Panas, and Bronis R. de Supinski

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,dquinlan,panas2,desupinski1}@llnl.gov

Abstract. OpenMP is a popular and evolving programming model for shared-memory platforms. It relies on compilers to target modern hardware architectures for optimal performance. A variety of extensible and robust research compilers are key to OpenMP's sustainable success in the future. In this paper, we present our efforts to build an OpenMP 3.0 research compiler for C, C++, and Fortran using the ROSE source-to-source compiler framework. Our goal is to support OpenMP research for ourselves and others. We have extended ROSE's internal representation to handle all OpenMP 3.0 constructs, thus facilitating experimenting with them. Since OpenMP research is often complicated by the tight coupling of the compiler translation and the runtime system, we present a set of rules to define a common OpenMP runtime library (XOMP) on top of multiple runtime libraries. These rules additionally define how to build a set of translations targeting XOMP. Our work demonstrates how to reuse OpenMP translations across different runtime libraries. This work simplifies OpenMP research by decoupling the problematic dependence between the compiler translations and the runtime libraries. We present an evaluation of our work by demonstrating an analysis tool for OpenMP correctness. We also show how XOMP can be defined using both GOMP and Omni. Our comparative performance results against other OpenMP compilers demonstrate that our flexible runtime support does not incur additional overhead.

1 Introduction

OpenMP [1] is a popular parallel programming model for shared memory platforms. By providing a set of compiler directives, user level runtime routines and environment variables, it allows programmers to express parallelization opportunities and strategies on top of existing programming languages like C/C++ and Fortran. As a proliferation of new hardware architectures becomes available, OpenMP has become a rapidly evolving programming model; numerous improvements are being proposed to broaden the range of hardware architectures that it

* This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 15–28, 2010.

can accommodate. A variety of robust and extensible compiler implementations are the key to OpenMP's sustainable success in the future since an OpenMP compiler should deliver portable performance. Open source OpenMP compilers permit active research for this rapidly evolving programming model.

Developed at Lawrence Livermore National Laboratory, the ROSE compiler [2] is an open source compiler infrastructure to build source-to-source program translation and analysis tools for large-scale C/C++ and Fortran applications. Given its stable support for multiple languages and user-friendly interface to build arbitrary translations, ROSE is particularly well suited to build reference implementations for parallel programming languages and extensions. It also enables average users to create customized analysis and transformation tools for parallel applications. In this paper, we present our efforts to build an OpenMP research compiler using ROSE. Our goal is to support OpenMP research for ourselves and others. For example, we have extended ROSE's internal representation to represent the latest OpenMP 3.0 constructs faithfully and to facilitate their manipulation; allowing the construction of custom OpenMP analysis tools.

More generally, the tight coupling of the compiler translations and the runtime system upon which they depend often complicate OpenMP research. Changing the existing compiler translations to utilize a new runtime library (RTL) usually requires significant effort. Conversely, changing the RTL when new features require support from compiler translations can be difficult. This tight coupling impedes research work on the OpenMP programing model. We seek to use ROSE as a testbed to decouple compiler translations from the OpenMP runtime libraries. We have designed and developed a common RTL interface and a set of corresponding compiler translations within ROSE. As a preliminary evaluation, we demonstrate an OpenMP analysis tool built using ROSE and the initial performance results of ROSE's OpenMP implementation targeting the OpenMP RTLs of both GCC 4.4 and Omni [3] 1.6.

The remainder of this paper is organized as follows. In the next section, we introduce the design goal of ROSE and its major features as a source-to-source compiler framework. Section 3 describes the OpenMP support within ROSE, including internal representation, a common RTL, and translation support. Section 4 presents a preliminary evaluation of ROSE's OpenMP support. We discuss related work in Section 5 while we present our conclusions and discuss future work in Section 6.

2 The ROSE Compiler

ROSE [4,2] is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C/C++ and Fortran applications. It also has increasing support for parallel applications using OpenMP, UPC and MPI. Similar to other source-to-source compilers, ROSE consists of frontends, a midend, and a backend, along with a set of analyses and

optimizations. Essentially, it provides an object-oriented intermediate representation (IR) with a set of analysis and transformation interfaces allowing users to build translators, analyzers, optimizers, and specialized tools quickly. The intended users of ROSE are experienced compiler researchers as well as library and tool developers who may have minimal compiler experience.

A representative translator built using ROSE works as follows (shown in Fig. 1). ROSE uses the EDG [5] front-end to parse C (also UPC) and C++ applications. Language support for Fortran 2003 (and earlier versions) is based on the open source Open Fortran Parser (OFP) [6]. ROSE converts the intermediate representations (IRs) produced by the front-ends into an intuitive, object-oriented abstract syntax tree (AST). The AST exposes interface functions to support transformations, optimizations, and analyses via simple function calls. Our object oriented AST includes analysis support for call graphs, control flow, data flow (e.g., live variables, def-use chain, reaching definition, and alias analysis), class hierarchies, data dependence and system dependence. Representative program optimization and translation interfaces cover partial redundancy elimination, constant folding, inlining, outlining [7], and loop transformations [8]. The ROSE AST also allows user-defined data to be attached to any node through a mechanism called persistent attributes as a way to extend the IR to store additional information. The ROSE backend generates source code in the original source language from the transformed AST, with all original comments and C preprocessor control structures preserved. Finally, ROSE can call a vendor compiler to continue the compilation of the generated (transformed) source code; generating a final executable. ROSE is released under a BSD-style license and is portable to Linux and Mac OS X on IA-32 and x86-64 platforms.

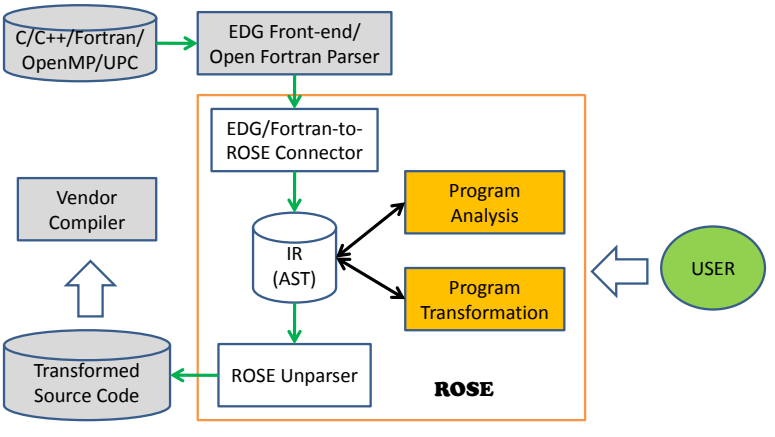


Fig. 1. A source-to-source translator built using ROSE

3 OpenMP Support in ROSE

As Fig. 2 shows, ROSE supports parsing OpenMP 3.0 constructs for C/C++ and Fortran[1], creating their internal representation as part of the AST, and regenerating source code from the AST. Additional support includes a set of translations targeting multiple OpenMP 2.5/3.0 RTLs based on XOMP, our common OpenMP RTL that abstracts the details of any specific RTL (such as GCC's OpenMP RTL GOMP [9] and the Omni [3] compiler's RTL). An automatic parallelization module is also available in ROSE [10].

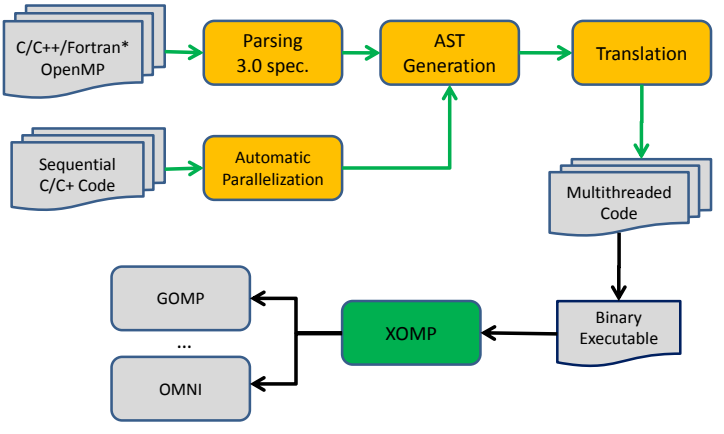


Fig. 2. OpenMP support in ROSE

3.1 Parsing and Representing OpenMP

Neither EDG (version 4.0 or earlier) nor OFP recognize OpenMP constructs. The raw directive strings exist in the ROSE AST as pragma strings for C/C++ and source comments for Fortran. Thus, we had to develop two OpenMP 3.0 directive parsers within ROSE, one for C/C++ and the other for Fortran. This, however, has significant advantages for users since they can easily change our parsers to test new OpenMP extensions without dealing with EDG or OFP.

ROSE's OpenMP parsers process OpenMP directive strings and generate a set of data structures representing OpenMP constructs. These data structures are attached to relevant AST nodes as persistent AST attributes. Using persistent AST attributes as the output of the parsers simplifies the work for parsing since we only make minimal changes to the existing ROSE AST. In fact, this light-weight representation for OpenMP is also used as the output of ROSE's automatic parallelization module [10]. As a result, the remaining OpenMP-related processing can work on the same input generated either from user-defined OpenMP programs or automatically generated OpenMP codes.

[1] Translation of Fortran OpenMP applications is still ongoing work.

After that, a conversion phase converts the ROSE AST with persistent attributes for OpenMP into an AST with OpenMP-specific AST nodes, which include statement-style nodes for OpenMP directives and supporting nodes (with file location information) for OpenMP clauses. Compared to the auxiliary persistent attributes attached to AST nodes, the newly-introduced AST nodes for OpenMP directives and clauses are inherently part of the ROSE AST. Thus, we can directly resuse most existing AST traversal, query, scope comparison, and other manipulation interfaces developed within ROSE to manipulate OpenMP nodes. For instance, a regular AST traversal is able to access all variables used within the AST node for an OpenMP clause with a variable list. This significantly simplifies the analysis and translation of OpenMP programs.

3.2 OpenMP Translation and Runtime Support

An OpenMP implementation must translate OpenMP applications into multithreaded code with calls to a supporting runtime library. To offer maximal freedom and optimization opportunities to OpenMP implementations, the OpenMP specification does not mandate the interface between a compiler and a runtime library. The implementation must decide what work to defer to the runtime library and how the compiler translation interacts with the library. Therefore, an OpenMP compiler's translation is traditionally tightly coupled with a given runtime library's interface. It is often a major effort to change the existing compiler translation to utilize a new runtime library. However, different runtime library choices and changes in the interactions between the compiler can significantly impact OpenMP performance. Thus, it would be especially desirable for an OpenMP research compiler to support multiple OpenMP runtime libraries.

Fortunately, although OpenMP runtime library interfaces vary, they usually include many similar or overlapped runtime library functions. For example, most portable OpenMP runtime libraries rely on the Pthreads API to create and to manipulate threads. Such a library usually provides a function that accepts a function pointer and a parameter to start multiple threads. Similalry, the OpenMP specification prescriptively defines some aspects of loop scheduling policies so runtime support for them often significantly overlaps.

We have introduced a common OpenMP RTL, XOMP, so that ROSE requires minimal changes to support multiple OpenMP RTLs. Depending on the similarity among RTLs, we use three rules in order to define XOMP and the corresponding compiler transformations.

- **Rule 1.** Target RTLs have some functions with similar functionalities. Those functions often differ by names and/or parameter lists. For each of the functions, we define a common function name and a union set of parameters in XOMP. The implementation of the common function will handle possible type conversion, parameter dispatch, inclusions/exclusions of functionality (to compensate for minor differences) before calling different the target RTL internally. By doing this, we can use one translation targeting XOMP's functions across multiple RTLs.

- **Rule 2.** Compared to other RTLs, a target RTL, libA, has an extra function, funcA().
  1. libA needs to call funcA() explicitly while other libraries do not have a similar need or meet the need transparently. We define an interface function in XOMP for funcA(). The implementation of the XOMP function is conditional based on the target RTL, either calling funcA() for libA or doing nothing for all others. Compiler translation targets the same XOMP interface as if all RTLs had the explicit need.
  2. funcA() implements some common functionality that is indeed suitable to be put into an RTL. Other libraries lack the similar support and rely on compiler translation too much. We define an XOMP function for the common functionality. The XOMP function either calls funcA() for libA or implements the functionality that is absent in other RTLs. Compiler translation targets the XOMP function.
  3. funcA() implements some functionality that is better suited to direct implementation by compiler translation. We develop the compiler translation to generate statements to implement the functionality without leveraging any runtime support. Still, the compiler translation can work with all RTLs.
- **Rule 3.** Occasionally, none of the above options may apply nicely. For example, the translation methods and the corresponding runtime support for an OpenMP construct can be dramatically different. In this case, we expose all the runtime functions in XOMP and have different translations for different XOMP support depending on the choice of implementation.

Finally, OpenMP translations share many similar tasks regardless of their target RTLs. These tasks include generating an outlined function to be passed to each thread, variable handling for shared and private data, and replacing directives with a function call. We have developed a set of AST transformations to support these common tasks. For example, the ROSE outliner [7] is a general-purpose tool to extract code portions from both C and C++ to create functions. It automatically handles variable passing according to variable scope and use information.

3.3 Translation Algorithm

We use the following translation algorithm for each input source file that uses OpenMP:

1. Use a top-down AST traversal to make implicit data-sharing attributes explicit, including implicit **private** loop index variables for loop constructs and implicit **firstprivate** variables for task constructs.
2. Use a bottom-up AST traversal to locate OpenMP nodes and to perform necessary translations.
   (a) Handle variables if they are listed within any of **private**, **firstprivate**, **lastprivate** and **reduction** clauses of a node.

(b) For (**omp parallel**) and (**omp task**) constructs, generate outlined functions as tasks and replace the original code block with XOMP runtime calls.
(c) For loop constructs, normalize target loops and generate code to calculate iteration chunks for each thread, with the help from XOMP loop scheduling functions.
(d) Translation for other constructs, such as **barrier**, **single**, and **critical**, are relatively straightforward [11].

Our algorithm handles variables with OpenMP data-sharing attributes in a separate phase before other translation activities. Thus, we eliminate OpenMP semantics from a code segment as much as possible so the general-purpose ROSE outliner can easily handle the code segment. Combined OpenMP variable handling and outlining would otherwise force us to tweak the outliner to handle OpenMP data-sharing variables specially during outlining.

3.4 Examples

We take the GCC 4.4.1's GOMP [9] and Omni Compiler [3] (v1.6) RTLs as two examples to demonstrate the definition of XOMP and the corresponding reusable compiler translations. GOMP is a widely available OpenMP runtime library and has recently added support for the task features of OpenMP 3.0. The Omni compiler is a classic reference research compiler for OpenMP 2.0/2.5 features. Supporting these two representative RTLs within a single compiler is a good indication of extensibility of a research compiler.

Fig. 3 and Fig. 4 give an example OpenMP program that uses tasks and ROSE's OpenMP translation that targets XOMP. ROSE uses a bottom-up traversal to find OpenMP parallel and task nodes and generates three outlined functions with the help from the outliner. These outlined functions are passed to either XOMP_parallel_start() or XOMP_task() to start multithreaded execution.

Some XOMP functions, such as XOMP_parallel_start(), XOMP_barrier() and XOMP_single(), are defined based on Rule 1 as common interfaces on top of both GOMP and Omni's interfaces. Rule 2.1 applies to XOMP_init() and XOMP_terminate(), which are introduced by Omni to initialize and to terminate runtime support explicitly while GOMP does not need them. In another case, GOMP does not provide runtime support for some simple static scheduling while Omni does. We decided to use Rule 2.3, letting the translation generate statements calculating loop chunks for each thread and totally ignore any runtime support. Rule 3 applies to the implementation for **threadprivate**. GCC uses Thread-Local Storage (TLS) to implement **threadprivate** variables. The corresponding translation is simple: mostly by adding the keyword __thread in front of the original declaration for a variable declared as **threadprivate**. On the other hand, Omni uses heap storage to manage threadprivate variables and relies on more complex translation and runtime support to initialize and to access the right heap location as a private storage for each thread. These two implementations represent two common methods to support **threadprivate** that each has well-known advantages and disadvantages. As a result, we decided to support

```
1  int main ()
2  {
3  #pragma omp parallel
4    {
5  #pragma omp single
6      {
7        int i;
8  #pragma omp task untied
9        {
10         for (i = 0; i < 5000; i++)
11         {
12 #pragma omp task if(1)
13           process (item[i]);
14         }
15       }
16     }
17   }
18   return 0;
19 }
```

Fig. 3. An example using tasks

both methods and to use different translation and/or runtime support conditionally depending on the choice of the final target RTL. XOMP_task is an exceptional case since Omni does not have corresponding support and we defined it based on GOMP's interface. In summary, less than 20% of the XOMP functions are defined using Rule 3. This means that more than 80% of the OpenMP translation can be reused across multiple RTLs.

Leveraging ROSE's robust C++ support, we are also able to implement OpenMP translation for C++ applications. Fig. 6 shows the translation result of an example C++ program shown in Fig. 5. The ROSE outliner supports generating an outlined function with C-bindings at global scope from a code segment within a C++ member function. This binding choice is helpful since the thread handling functions of most OpenMP RTLs expect a pointer to a C function, not a C++ one. The outlined function at line 22 is also declared as a friend (at line 11) in the host class to access all class members legally.

4 Evaluation

We evaluate ROSE's support for both OpenMP analysis and translation.

4.1 OpenMP Analysis

We have used ROSE to build a simple analysis tool that can detect a common mistake of using OpenMP locks. As Fig. 7 shows, a lock variable (at line 3) is declared within a parallel region and then used within that same parallel region, which is incorrect since a lock must be **shared** to be effective. A locally declared lock is **private** to each thread.

Fig. 8 shows the ROSE AST analysis code (slightly simplified) that can find this error in using locks. Programmers only need to create a class(OmpPrivateLock) by inheriting a builtin AST traverse class in ROSE and to provide a visitor

```
1  #include "libxomp.h"
2  struct OUT__1__1527___data {int i;};
3  struct OUT__2__1527___data {int i;};
4
5  static void OUT__1__1527__(void *__out_argv)
6  {
7    int i = (int )(((struct OUT__1__1527___data *)__out_argv) -> i);
8    int _p_i = i;
9    process((item[_p_i]));
10 }
11
12 static void OUT__2__1527__(void *__out_argv)
13 {
14   int i = (int )(((struct OUT__2__1527___data *)__out_argv) -> i);
15   int _p_i = i;
16   for (_p_i = 0; _p_i < 5000; _p_i++) {
17     struct OUT__1__1527___data __out_argv1__1527__;
18     __out_argv1__1527__.i = _p_i;
19     /* void XOMP_task (
20      * void (*fn) (void *), void *data, void (*cpyfn) (void *, void *),
21      * long arg_size, long arg_align, bool if_clause, bool untied )*/
22     XOMP_task(OUT__1__1527__,&__out_argv1__1527__,0,4,4,1,0);
23   }
24 }
25
26 static void OUT__3__1527__(void *__out_argv)
27 {
28   if (XOMP_single()) {
29     int i;
30     struct OUT__2__1527___data __out_argv2__1527__;
31     __out_argv2__1527__.i = i;
32     XOMP_task(OUT__2__1527__,&__out_argv2__1527__,0,4,4,1,1);
33   }
34   XOMP_barrier();
35 }
36
37 int main(int argc,int argv)
38 {
39   int status = 0;
40   XOMP_init(argc,argv);
41   /* void XOMP_parallel_start (
42    * void (*func) (void *), void *data, unsigned num_threads )*/
43   XOMP_parallel_start(OUT__3__1527__,0,0);
44   XOMP_parallel_end();
45   XOMP_terminate(status);
46   return 0;
47 }
```

Fig. 4. Translated example using tasks

function implementation. The traversal visits all AST nodes to find a use of an OpenMP lock within any of OpenMP lock routines (line 4-13). The code then detects if the use of the lock is lexically enclosed inside a parallel region (line 16-18) and if the declaration of the lock is also inside the same parallel region (line 21-22). The statement style OpenMP node(SgOmpParallelStatement) for a parallel region enables users to directly reuse AST interface functions, such as the function to find a lexically enclosing node of a given type (SageInterface::getEnclosingNode<ParentType>(node)) and another function to tell if a node is another node's ancestor (SageInterface::isAncestor(a_node, c_node)). This

```
1  class A
2  {
3    private:
4      int i;
5    public:
6      void pararun()
7      {
8  #pragma omp parallel
9        {
10 #pragma omp critical
11         cout<<"i= "<< i <<endl;
12       }
13     }
14 };
```

Fig. 5. A C++ example

```
1  #include "libxomp.h"
2  struct OUT__1__1527___data {  void *this__ptr___p; };
3  static void OUT__1__1527__(void *__out_argv);
4  static void *xomp_critical_user_;
5
6  class A
7  {
8    private:
9      int i;
10   public:
11     friend void ::OUT__1__1527__(void *__out_argv);
12     void pararun()
13     {
14       class A *this__ptr__ = this;
15       struct OUT__1__1527___data __out_argv1__1527__;
16       __out_argv1__1527__.this__ptr___p = (void *)this__ptr__;
17       XOMP_parallel_start(OUT__1__1527__,&__out_argv1__1527__,0);
18       XOMP_parallel_end();
19     }
20 };
21
22 static void OUT__1__1527__(void *__out_argv)
23 {
24   class A *this__ptr__ =
25        (class A *)(((struct OUT__1__1527___data *)__out_argv) -> this__ptr___p);
26   XOMP_critical_start(&xomp_critical_user_);
27   std::cout<<"i= "<<( *this__ptr__).i<<std::endl;
28   XOMP_critical_end(&xomp_critical_user_);
29 }
```

Fig. 6. Translated C++ example

```
1  #pragma omp parallel
2    {
3      omp_lock_t lck;
4      omp_set_lock(&lck);
5      printf("Thread = %d\n", omp_get_thread_num());
6      omp_unset_lock(&lck);
7    }
```

Fig. 7. Using a private lock

example demonstrates that writing analysis tools using ROSE is straightforward since OpenMP constructs are represented as nodes that are inherently part of the ROSE AST.

```
1  void OmpPrivateLock::visit(SgNode* node)
2  {
3    //1. Find an OpenMP lock routine
4    SgFunctionCallExp * func_call = isSgFunctionCallExp(node);
5    if (!func_call) return;
6    std::string f_name = func_call->get_name();
7    if (f_name != "omp_unset_lock" && f_name != "omp_set_lock"
8        && f_name != "omp_test_lock")     return;
9
10   //2. Grab the only routine parameter as the use of a lock
11   std::vector<SgVarRefExp*> exp_vec =
12       SageInterface::querySubTree<SgVarRefExp>(func_call, V_SgVarRefExp);
13   ROSE_ASSERT(exp_vec.size() ==1);
14
15   //3. If the lock's use is inside a parallel region
16   SgOmpParallelStatement* lock_region =
17     SageInterface::getEnclosingNode<SgOmpParallelStatement>(exp_vec[0]);
18   if (lock_region)
19   {
20   //4. Check if the lock declaration is also inside the same region
21      SgVariableDeclaration* lock_decl = exp_vec[0]->get_declaration();
22     if (SageInterface::isAncestor(lock_region, lock_decl))
23       cerr<<"Found a private lock within a parallel region"<<endl;
24   }
25 }
```

Fig. 8. A ROSE-based tool to find private locks

4.2 OpenMP Translation

We have evaluated ROSE's OpenMP translations and the corresponding XOMP interface through a set of OpenMP benchmarks, including the NAS Parallel Benchmarks(NPB)[12] and the Barcelona OpenMP Task Suite (BOTS) [13], Those benchmarks have builtin correctness verification so they also test the correctness of our compiler implementations. We ran all experiments on a Dell T5400 workstation with dual processors and 8 GB of memory. Each of the processors is a 3.16 GHz quad-core Intel Xeon X5460 processor. We used several other OpenMP compilers in addition to ROSE, including GCC 4.4.1, Intel Compilers 11.1.059, and the Mercurium 1.3.3 compiler with the Nanos 4.1.4 runtime. We used GCC 4.4.1 as the backend compiler for all source-to-source implementations. We used compiler option *-O3* whenever possible.

Fig. 9 shows the speedup of a subset of NPB (V 2.3 C version [14]) and BOTS V 1.0 using up to 8 threads by different compiler/runtime configurations. Results for the remaining benchmarks had similar patterns and are not shown for brevity. ROSE-Omni's speedup for the BOTS benchmarks (NQUEEN, SORT, and STRASSEN) is not available since the Omni runtime library does not support OpenMP tasking. In general, all implementations had comparable performance. ROSE's source-to-source translation and extra layer of runtime support do not incur any significant performance overhead compared to other compilers.

5 Related Work

Some other OpenMP research compilers exist. Representative examples include Omni [3], OdinMP [15] and OpenUH [11]. Most research compilers adopt the

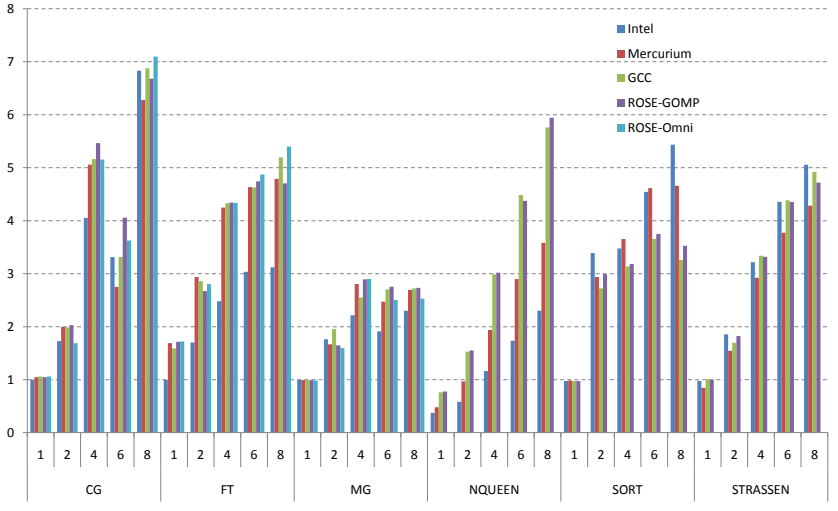


Fig. 9. Speedup of some NPB 2.3 and BOTS 1.0 benchmarks

source-to-source translation approach. Based on Open64, OpenUH supports both source-to-source translation and generating the final binary code by itself. The Nanos Mercurium compiler [16] is another source-to-source compiler aimed at fast prototyping for OpenMP. It was among the first to support OpenMP 3.0's task feature and was used to evaluate the expressiveness and flexibility of OpenMP task directives compared to using nested parallelism and Intel's taskqueues. More recently, Addison et. al. [17] presented the OpenMP 3.0 implementation in OpenUH [11] with an extended runtime system supporting tasking. However, the corresponding compiler translation was done manually, as reported in their paper. Leveraging GCC 4.4's runtime library, ROSE is one of the few OpenMP compilers supporting OpenMP 3.0. It might be the only OpenMP research compiler with stable C++ source-to-source support, although both OpenUH and Mercurium have a similar goal. Finally, ROSE's XOMP translation interface enables ROSE to implement translations targeting different RTLs quickly as demonstrated in this paper. Other compilers usually target only a single RTL.

6 Conclusion

In this paper, we have presented ROSE as an OpenMP 3.0 research compiler for C/C++ and Fortran. ROSE's OpenMP support includes extensions to ROSE's AST to represent OpenMP constructs, a common runtime support interface (XOMP), and a set of reusable translations that can target multiple OpenMP runtime libraries. An automatic parallelization module is also available in ROSE. Our AST representation for OpenMP is inherently part of the ROSE AST so

most existing AST manipulation, analysis, and transformation interface functions can be easily reused to handle OpenMP applications. Preliminary evaluation demonstrates that it is straightforward to write static analysis tools for OpenMP. Also, ROSE's OpenMP translation targeting two mainstream OpenMP RTLs has competitive performance compared to other OpenMP implementations. The latest ROSE OpenMP support has been released as part of the ROSE distribution (downloadable from our website [2]).

In the future, we plan to add the OpenMP Fortran support and to complete the OpenMP 3.0 implementation, such as loop collapse. We will build more static analysis tools to help users write correct OpenMP applications. With ROSE's unique C++ support, we are interested in exploring more C++-related issues within OpenMP. The introduction of explicit tasks in OpenMP 3.0 gives implementations and users more choices to optimize parameters related to tasks, such as the cut-off depth of tasks, tied or untied tasks, or task scheduling policies (including task aggregation granularity). We expect that empirical tuning can play an important role in finding the best OpenMP compilation and execution parameters for a given application on a particular platform. Finally, we especially welcome external collaborations using ROSE for research specific to the requirements of the OpenMP research community.

References

1. OpenMP Architecture Review Board: OpenMP application program interface, version 3.0 (2008), `http://www.openmp.org/mp-documents/spec30.pdf`
2. Quinlan, D.J., et al.: ROSE compiler project, `http://www.rosecompiler.org/`
3. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP compiler for an SMP cluster. In: The 1st European Workshop on OpenMP (EWOMP'99), September 1999, pp. 32–39 (1999)
4. Quinlan, D.: ROSE: Compiler support for object-oriented frameworks. In: Proceedings of Conference on Parallel Compilers, CPC (2000)
5. Edison Design Group: C++ Front End, `http://www.edg.com`
6. Rasmussen, C., et al.: Open Fortran Parser, `http://fortran-parser.sourceforge.net/`
7. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: The 22th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Newark, Delaware, USA (2009)
8. Yi, Q., Quinlan, D.: Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In: Eigenmann, R., Li, Z., Midkiff, S.P. (eds.) LCPC 2004. LNCS, vol. 3602, pp. 253–267. Springer, Heidelberg (2005)
9. GOMP - an OpenMP implementation for GCC (2005), `http://gcc.gnu.org/projects/gomp`
10. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 28–41. Springer, Heidelberg (2009)

11. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience 19(18), 2317–2332 (2007)
12. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)
13. Barcelona OpenMP task suite, `http://nanos.ac.upc.edu/content/barcelona-openmp-task-suite`
14. C version NPB 2.3 in OpenMP, `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html`
15. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12(12), 1193–1203 (2000)
16. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model, pp. 63–77 (2008)
17. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Open64 Workshop at CGO 2009 (2009)

Binding Nested OpenMP Programs on Hierarchical Memory Architectures*

Dirk Schmidl[1], Christian Terboven[1], Dieter an Mey[1], and Martin Bücker[2]

[1] JARA, RWTH Aachen University, Germany
Center for Computing and Communication
{schmidl,terboven,anmey}@rz.rwth-aachen.de
[2] JARA, RWTH Aachen University, Germany
Institute for Scientific Computing
buecker@sc.rwth-aachen.de

Abstract. In this work we discuss the performance problems of nested OpenMP programs concerning thread and data locality particularly on cc-NUMA architectures. We provide a user friendly solution and demonstrate its benefits by comparing the performance of some kernel benchmarks and some real-world applications with and without applying our affinity optimizations.

1 Introduction

The memory hierarchy of modern HPC architectures is becoming more and more complex. As of today, most processors employ three levels of caches and multiprocessor systems are revealing their non uniform memory access (cc-NUMA) behavior. In order to exploit the potential of these architectures fully, the programmer has to be aware of these characteristics. Shared-memory parallel programs have to be optimized for data-to-processor affinity.

There are some mechanisms helping the programmer to take care of data locality on cc-NUMA architectures for OpenMP programs [9]. However, problems arise if these mechanisms are applied to nested OpenMP programs, which are programs employing two or more OpenMP parallel regions, one nested inside the other. The main problem is the way in which compilers manage OpenMP threads for these programs. Current OpenMP runtime environments organize the threads in a thread pool. When a parallel region is encountered the team is formed by taking some threads out of the pool. Unfortunately, it is neither guaranteed by the standard nor usually the case that whenever an inner nested parallel region is encountered multiple times the team consists of the same threads. If the set of system threads an OpenMP team consists of changes, data affinity is lost.

For OpenMP programs employing only a single level of parallelism it is common practice to use the first touch mechanism provided by modern operating systems to

* This research is partially supported by the German Federal Ministry of Education and Research (BMBF) under the contract 03SF0326A "MeProRisk: Novel methods for exploration, development, and exploitation of geothermal reservoirs - a toolbox for prognosis and risk assessment."

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 29–42, 2010.

place the data next to the thread working on it. This is achieved by initializing the data in parallel, so that every thread initializes the data it is going to use later on. This is not applicable in the case of nested OpenMP because the set of system threads an inner team consists of is not persistent. We propose a set of strategies to be specified by the user advising the OpenMP runtime how threads in nested OpenMP programs have to be bound to preserve data locality. We provide a reference implementation in the form of a thread binding library. We also propose how to extend the OpenMP standard to offer this functionality.

This paper is structured as follows: Section 3 describes our affinity optimizations as offered by our binding library. Section 4 gives an overview of the hardware we used for evaluating our approach. We then demonstrate the feasibility of our approach using kernel benchmarks in Section 5 and two real-world applications arising from different scientific disciplines in Section 6. Finally, we draw our conclusions in section 7.

2 Related Work

There are different approaches in this field of research. [6] introduced the idea of subteams to manage the grouping of OpenMP threads. An extension to OpenMP is presented which forms subteams of threads and maps worksharing constructs on these subteams using the ONTHREADS clause. This idea allows to map different work packages on different subteams, thus it is an explicit alternative to nested parallel regions. It does not consider the mapping between threads and cores or processes to the hardware. The main difference to our approach is, that we want to propose an extension to the nesting concept in OpenMP, which handles the mapping between threads and the underlying hardware.

[1] and [11] both present a method to bind threads on a specific core of the hardware. In both approaches a group of processors can be assigned to a team of threads, using a new clause to extend OpenMP. In the first approach the groups must be specified using a special GROUPS clause. Afterwards the work can be assigned to the master of a group using an ONTO clause. The slave threads of a group can later on be used inside of a nested parallel construct, when the master spawns a new team. This approach is useful, when sections are used on the outer level, or when the number of threads to be used on this level is constant, because the groups are specified with a string at compile time. A varying number of threads on the outer level is not supported.

[11] uses a default processor group, which is a linear array of all processors in the system, but it is also possible to build more complex structures, which can represent nearly all hierarchical memory architectures. The programmer is responsible to assign work to these groups. The programmer can specify a start point and a stride to specify which processors to use. One downside of this approach is, that the programmer is required to have detailed hardware knowledge to achieve a good placement of threads. Especially for nested OpenMP programs finding the array slice to use for every team might become quite complicated.

A different approach is presented in ForestGOMP[10]. Here threads can be put together in groups called bubbles. These bubbles can have a hierarchical structure to describe a nesting relation. A scheduler called bubble sched is used to schedule the threads

to specific cores of the system. A thread stealing mechanism allows to change the mapping and threads can be migrated during a run if necessary. This mechanism is useful, when the load changes in a dynamic manner. The scheduler should take NUMA criteria into account for his scheduling decisions. A disadvantage of this approach is, that it is hard for the programmer to understand what the scheduler does.

In many cases the programmer might not have a detailed hardware knowledge, but usually detailed knowledge of the program. Out approach tries to abstract from the hardware details, while allowing the programmer to bring in some knowledge about the specific program. The strategies we ask the programmer to select from are platform-independent and are mapped to the actual machine at runtime. Furthermore we perform our investigations with different kernels and applications on a machine with a flat memory hierarchy on a cc-NUMA machine and on a special machine from the vendor ScaleMP to show that thread placement is of varying interest on different machines and that the interest will most likely grow in the near future.

3 Thread Binding Library

Exploiting thread-to-core binding facilities and the first-touch strategy of current operating systems is the standard approach to address data locality in shared-memory parallel programs [9]. Since for nested OpenMP parallel regions it is not guaranteed that any two active parallel regions with identical ancestry will be executed by the same set of system threads, this approach may fail. We provide a library that binds OpenMP threads on specific cores of the machine and ensures a persistent mapping of OpenMP threads to cores, even if the underlying system threads change.

With two or more nested levels of parallelism, possibly exhibiting different characteristics concerning core, cache and memory utilization, the selection of a core to bind a thread to has a significant influence on the application performance. We need detailed hardware knowledge in order to place threads on specific cores appropriately. For example it is necessary to know the number of sockets, cores and hardware-threads of a machine as well as information on the numbering scheme for threads used by the operating system. Many programmers do not have this knowledge, nor do they want to care about these details every time a new architecture is deployed. Our library detects these details automatically by examining the system architecture as will be described below. The user's task is reduced to specifying a *strategy* for thread placement and the library takes care of mapping the threads to specific cores, according to the given strategy. The concept of asking the programmer to specify a strategy that is related to certain application characteristics, as will be described below, intends to enable some level of performance portability as the strategies also may fit on a new machine; only the runtime has to be adapted. The binding library consists of the following three parts.

1. Hardware information: As described above, our approach requires detailed system information to process the mapping of threads to cores in the desired way. We provide two ways for the library to get this information. The first one is to query information about the system architecture and generate a tree structure with hardware information. This is the default way and is done automatically by the library. On Linux parsing

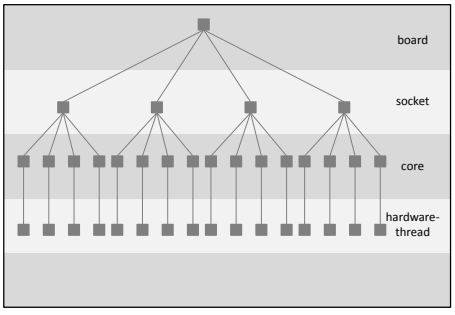


Fig. 1. Hardware tree of the Tigerton machine using `/proc/cpuinfo`

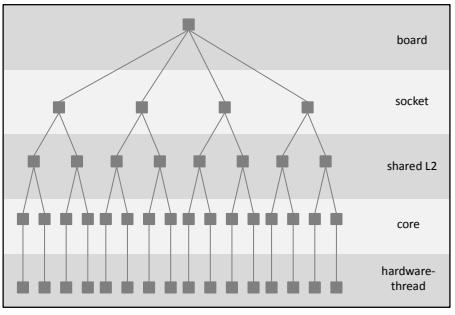


Fig. 2. Hardware tree of the Tigerton machine using a user-specified file

`/proc/cpuinfo` can distinguish between sockets, cores on a socket and hardware-threads running on a core. An example of such a tree structure is illustrated in Fig. 1. It represents the Tigerton machine used in our experiments throughout this article.

In many cases this information is sufficient to perform the placement, but for some special cases more detailed information may be advantageous. For example on the Tigerton machine, there are four cores on one socket with two cores sharing a single L2 cache. If the user wants the library to take this into account, he creates a file containing hardware information and sets the environment variable `OMP_CORE_LIST` to point to that file. This way a hardware tree for our library is created that recognizes cache sharing. The resulting tree, consisting of an additional level grouping together the cores sharing an L2 cache, is depicted in Fig. 2. Building the tree structure is carried out once at program startup. The corresponding overhead does not influence the total runtime significantly, in particular for large-scale applications.

2. Thread information: To support mapping the threads of nested OpenMP parallel regions to the hardware tree, the library needs to collect information about the complete thread hierarchy. This is done at program startup, avoiding too much overhead. A remapping at runtime is also possible by calling a suitable library function, as described below. The library supports the following strategies:

- **scatter:** Places the threads as far away from each other as possible. This is intended to maximize the memory bandwidth, but may lead to higher synchronization overhead.
- **compact:** Places the threads near each other. This minimizes synchronization times, but reduces the achievable memory bandwidth, especially on cc-NUMA architectures.
- **subcompact/subscatter:** These strategies behave like scatter and compact with one exception: In a nested parallel region, the new threads are placed nearer to their own master thread than to the master thread of another inner team. On cc-NUMA machines, this strategy aims at enforcing the team to run on the same socket as the master. So, the team might find shared data the master has initialized in its the local memory.

The number of threads to be used along with the strategy can be specified in two different ways:

1. Using the environment variable `OMP_NESTING_TEAM_SIZE`. If this variable is set the value is expected to be a list of the sizes of the teams to be used in the parallel regions encountered in the program flow. This approach incurs the restriction to use the same number of threads for all teams on the same nesting level. `OMP_NESTING_TEAM_SIZE=2,scatter,4,subcompact` means that, the outer parallel region is started with two threads that are placed according to the `scatter` strategy and both threads will start an inner team with four threads. For the inner teams the `subcompact` strategy will be used.
2. Using the function call `omp_set_nesting_info(s)`. This function expects a string `s` as an argument build in the same way as for `OMP_NESTING_TEAM_SIZE`. The function sets the strategy for the following parallel regions. If called in the serial part of the program the effect is the same as of `OMP_NESTING_TEAM_SIZE`. However, if it is called inside a parallel region, it changes the strategy for the calling thread (and all its descendents). Using this functionality it is also possible to have different teams of different sizes on the same nesting level. If the number of threads used inside the parallel regions changes over runtime, the environment variable is not sufficient again, instead `omp_set_nesting_info(s)` has to be used and the programmer has to specify the strategy to use for the next parallel regions before they are encountered.

The library takes this information to map the threads to cores using the hardware information. Since the intended core for every thread is stored with the thread hierarchy, the mapping has to be done only once at program startup and every time the function `omp_set_nesting_info(s)` is called, but not every time a parallel region is entered.

3. Thread placement: As described before, the problem of nested OpenMP is that it cannot be predicted which system threads are used to build a team. But only system threads can be bound to processors. The only way for a library (other than the OpenMP runtime) to find out which system threads are used inside a parallel region is to start the parallel region. Afterwards the threads can still be bound according to the strategy. This step has to be repeated every time a parallel region is entered. In order to enable this functionality a function call to our library has to occur at the beginning of every parallel region to find out which threads are used. It will also perform the look-up in the thread hierarchy table and perform the binding.

Instead of placing the burden of modifying all parallel regions in a given code on the programmer, we use the Opari tool [8] to instrument the source code. It inserts function calls before and after OpenMP constructs. It is designed and used for performance measurement tools. It inserts the function `pomp_parallel_begin` at the beginning of every parallel region and we use this function call to transfer the program flow to our library. Thereby the only elements of our approach visible to the programmer are the environment variable and API functions, which we propose to add to the OpenMP standard.

4 Computing Platforms for Experiments

We used three different platforms for our experiments, all running CentOS Linux with kernel 2.6.18 on the Tigerton and Barcelona and kernel 2.6.21 on the ScaleMP:

Tigerton (Fujitsu-Siemens RX600): This machine is equipped with 4 Intel Xeon X7350 processors with a clock rate of 2.93 GHz and 64 GB of memory. All processors access the whole main memory via Intel's north bridge, therefore this is an SMP system with a flat memory architecture.

Barcelona (IBM eServer LS42): This is a 4 socket machine as well, equipped with AMD Opteron 8356 processors. Every processor runs at a clock rate of 2.3 GHz and is connected to 8 GB of local memory. Remote access to the memory of other processors is accomplished via the Hyper-Transport interconnect. This machine offers a cc-NUMA architecture.

ScaleMP: From a hardware point of view, this machine is not a single node, but a small Infiniband cluster. All 13 nodes are equipped with two Intel Xeon E5420 processors (2.5 GHz) and 16 GB of memory. What makes this machine interesting for shared-memory programs is the vSMP Foundation software. This software implements a cache coherence protocol over the InfiniBand network and makes the machine look like a single shared memory machine with 104 cores and 170 GB of memory, from the perspective of a standard Linux operating system. In total the machine has 208 GB of memory, but roughly 38 GB are used for caching and prefetching mechanisms by the vSMP Foundation software and are not visible to the user. Every processor in the system can access all the memory of other boards, but of course getting data from other boards generates some overhead and is more expensive than working on local data. Therefore, we consider this to be a cc-NUMA machine as well, but with a much higher NUMA ratio (ratio of remote latency to local latency) than the Barcelona machine.

5 Kernel Benchmarks

To make sure that the library works as designed, we first used some kernel benchmarks for verification purposes. In summary, we find that placing threads apart over the respective machine increases memory bandwidth as well as synchronization time, independent of whether the threads belong to an outer or inner parallel level.

5.1 Nested Stream Benchmark

As the first test program, we choose a Stream-like benchmark to measure the obtained memory bandwidth of a system. We start with the original Stream [7] benchmark and change the program to use nested OpenMP on two levels. The original benchmark with one level of parallelism uses three arrays `a`, `b` and `c`. The initialization of the data is done in parallel using the same loop schedule to achieve a good placement of the data. On these arrays, a DAXPY operation is computed in parallel. The code for this operation is shown in code 1. The time to compute this vector operation is measured and the memory bandwidth is derived. In order to use nested OpenMP, we change the

Code 1. Parallel loop computing the DAXPY operation in the Stream benchmark.

```
C$omp do schedule(static)
          DO 60 j = 1,n
               a(j) = b(j) + scalar*c(j)
   60     CONTINUE
C$omp end do
```

benchmark to start an outer team of threads and all these threads run the original Stream benchmark. So every inner team gets its own arrays `a`, `b` and `c`, but they are shared inside the inner teams. We use the `scatter` strategy to distribute the threads of the outer parallel region over different sockets/boards of the machines. The inner threads are bound in such a way that they fill up the sockets/boards of the master thread using the `subcompact` strategy. Using this strategy we expected to get the best memory bandwidth, especially on the cc-NUMA machines.

Table 1 shows the achieved memory bandwidth on the three different machines. The *bound* values denote the measurements employing our library with the aforementioned strategy settings; the *unbound* values were obtained by deferring control of the thread placement solely to the operating system and the OpenMP runtime system. The columns labeled by `oxi` denote a nested parallelization strategy with `o` threads in the outer level and `i` threads in the inner level, resulting in a total number of `o*i` threads. In the first column, where just one thread is used, it is no surprise that the memory bandwidth does not depend on whether binding is used or not.

With 1x4 threads, we see the first differences between the machines. On the cc-NUMA machine (Barcelona) we can profit from binding as expected, but on the SMP machine (Tigerton) the memory bandwidth drops down in the bound case. Binding with our strategy makes all four inner threads run on the same socket on both machines. On the Barcelona machine, this is an advantage, because all the data is in the memory of this socket and so we just have local memory accesses. On the Tigerton machine, there is just one path to the memory. One socket cannot consume the total memory bandwidth on this machine, thus the unbound case profits from using four sockets.

Table 1. Memory bandwidth in GB/s of the nested Stream benchmark

threads	1x1	1x4	4x1	4x4	6x1	6x4	13x1	13x4
Barcelona								
unbound	4.4	4.9	15.0	10.7				
bound	4.4	7.6	15.8	13.1				
Tigerton								
unbound	2.3	6.0	4.8	8.7				
bound	2.3	3.0	8.2	8.5				
ScaleMP								
unbound	3.8	10.7	11.2	1.7	9.0	1.6	3.4	2.4
bound	3.8	5.9	14.4	18.8	20.4	15.8	43.0	27.8

In the case with four outer and one inner threads, we have the same total number of threads, but now every outer thread initializes its own arrays `a`, `b` and `c`. On the Barcelona machine this leads to a much higher total memory bandwidth, because now the memory of all four sockets is used. Even in the unbound case, we see nearly the same bandwidth. It seems that the scheduler is smart enough to use all the four sockets. On the Tigerton machine, the bound case delivers good memory bandwidth as well, but in the unbound case the result is worse. We suspect that the scheduler moves threads around.

Using four inner and four outer threads (all cores are busy on the Barcelona and Tigerton machine), probably the most relevant case for most applications, we can see on the Barcelona machine that binding clearly pays off. We reach a memory bandwidth of 13.1 GB/s in the bound and only 10.7 GB/s in the unbound case, an improvement of about 25%. On the Tigerton machine it does not matter whether we use binding or not, since it is an SMP system with a flat memory architecture.

On the ScaleMP machine the behavior for small numbers of threads looks similar to the Tigerton machine: for 1x4 threads the unbound case is better than the bound one and with 4x1 threads the bound case is better. This is because one board of the machine is an SMP system and Linux schedules up to eight threads all on the first board. For a larger number of threads the behavior resembles the Barcelona cc-NUMA machine: The bound cases outperform the unbound cases as soon as more than four threads are used. When the whole machine is used the bound case is more than 10 times faster than the unbound one. Another interesting point is, that we get a better result for 13x1 and 6x1 than for 13x4 and 6x4. Normaly we would expect that we can profit from more than one thread per board, but it seems that the overhead of the vSMP software grows with the number of total threads and therefore the maximum memory bandwidth is reached for one thread per board for a larger number of boards.

5.2 Nested EPCC Syncbench Benchmark

The second kernel benchmark is a modification of the `syncbench` benchmark from the EPCC microbenchmarks [2]. It measures the overhead of OpenMP synchronisation constructs. We want to investigate the overhead of nested OpenMP constructs, so we modify the benchmark in the same way we modify the stream benchmark. Our benchmark opens a parallel region and starts the EPCC syncbench for each of the inner teams.

Our assumption is that synchronization is more efficient when the inner teams run close together. When all threads of an inner team run on the same socket for example, they can communicate using a shared cache. Table 2 shows the overhead of a few OpenMP constructs in the nested case. Again we use the `scatter` strategy on the outer level and the `subcompact` strategy on the inner level.

For 1x2 threads, we see slight differences on the different machines. On the Tigerton and the ScaleMP machine, we can reduce the overhead of all constructs significantly by binding the threads closely together using the `subcompact` strategy. On the Barcelona machine, binding increases the measured overhead slightly. The reason for this behavior is that the Intel Xeon processors (of both, Tigerton and ScaleMP) have a shared L2 cache for two of the cores. When the team runs on these two cores, the communication is very fast. On the Barcelona machines the L2 cache is not shared, but the L3 cache is.

Table 2. Overhead of nested OpenMP constructs in μs

	parallel	barrier	reduction	parallel	barrier	reduction
Barcelona	1 x 2			4 x 4		
unbound	19.25	18.12	19.80	117.90	100.27	119.35
bound	23.53	20.97	24.14	70.05	69.26	69.29
Tigerton	1 x 2			4 x 4		
unbound	23.88	20.84	24.17	74.15	54.90	77.00
bound	9.48	7.35	9.77	58.96	34.75	58.24
ScaleMP	1 x 2			2 x 8		
unbound	63.53	65.00	42.74	2655.09	2197.42	2642.03
bound	34.11	33.47	41.99	425.63	323.71	444.77

Communication using the L3 cache is slower resulting in a higher syncronisation time compared to the Xeon-based systems.

When 4x4 threads are used, the results look quite different. We do not see the cache effects anymore, because only two threads can share an L2 cache and all four threads are involved in a synchronization operation. But we see that binding has a positive effect on all machines. When all threads of the inner teams run on one socket on the Barcelona and Tigerton machine and on one board for the ScaleMP machine, the communication is faster. In the unbound case the system scheduler seems to schedule them over the whole machine, this causes higher synchronisation overhead. On the ScaleMP machine, where the InfiniBand network is used to communicate between different boards, the overhead is significant. A barrier operation for example takes 2197.4 μs when the threads are scheduled by the operating system scheduler and just 323.7 μs when the binding is done in an appropriate way. On the other machines, the difference is smaller, but still noticeable and relevant as synchronization constructs limit the scalability of many applications.

6 Applications

With the kernel benchmarks, we have shown that our binding improves the performance of synchronization constructs and the achievable memory bandwidth in most well understood cases. To analyze the influence on real-world applications we consider two codes with different performance characteristics.

6.1 TFS

The first code is a multiblock Navier-Stokes solver called TFS [4]. It is developed by the Institute of Aerodynamics at RWTH Aachen University. The package is used to simulate the airflow through a human nose. The code consists of 17,000 lines of Fortran code and it is parallelized using OpenMP on two levels [5]. At the outer level the code is parallelized by a domain decomposition approach characterized by different blocks of the computational grid. Depending on the dataset the number of threads that can be used

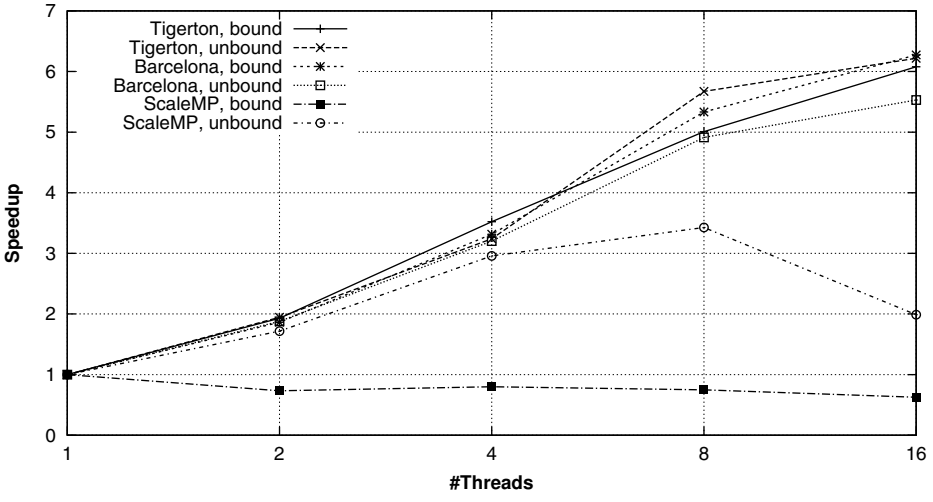


Fig. 3. Best speedup for different total thread numbers of the TFS code

is limited, because the data cannot be split into an arbitrary number of blocks. The nose dataset provides 16 blocks of different sizes. On the inner nested level of parallelization the do-loops inside the solver are parallelized, allowing the use of more threads than there are blocks in the dataset.

We test the application on the three machines with and without binding and compare the results. We use a `scatter` binding strategy on the outer level and a `subscatter` strategy on the inner one. This strategy should optimize the total memory bandwidth and reduce the synchronization time of the inner teams. Fig. 3 shows the best results for a fixed number of total threads. This means, for example, the speedup for 4 threads is actually the maximum of the speedups obtained from the cases 4x1, 2x2, and 4x1.

We see different behaviors on the three different machines. On the Tigerton machine the application shows a speedup of about 6 using 16 threads. It does not make a big difference whether binding is used or not. On the Barcelona machine we see a similar speedup in the bound case, but here the unbound case is worse. It is just around 5.5. So, binding results in a performance improvement of roughly 15 % on this machine. On the ScaleMP machine, the results look completely different. The unbound version of the code scales up to 8 threads and then drops down. Taking a closer look at the machine while the program is executing, we see that 8 threads still run on one board and if more threads are used, the threads get automatically migrated to other boards, resulting in a performance loss. In the bound case we force the threads to run on different boards. Because of that we do not gain any advantage from the parallelization. We could change the strategy for our binding library to stay on one board as long as we do not use more than eight threads, but in doing so we do not employ the ScaleMP advantages.

In summary, we assess that the TFS code is not well-suited for the ScaleMP machine. With and without binding we cannot profit from the special characteristics of the system. On the Tigerton machine binding is not advantageous, but on the Barcelona machine we can improve the performance by 15%.

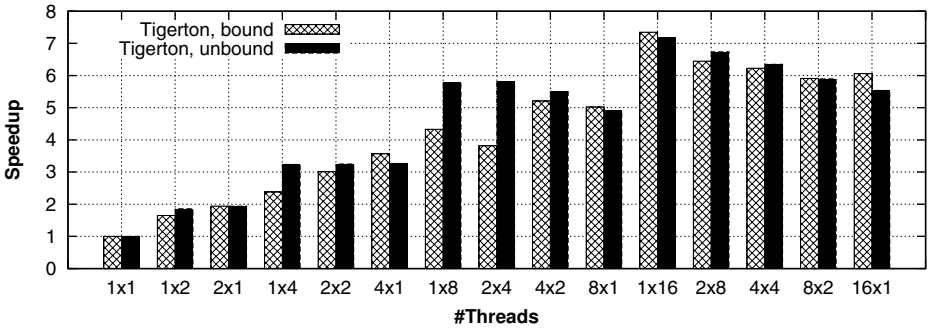


Fig. 4. Best speedup for the SHEMAT-Suite code on the Tigerton machine

6.2 SHEMAT-Suite

The second application is called SHEMAT-Suite, the successor of the code named SHEMAT [3] and is still being developed by the Institute for Applied Geophysics and Geothermal Energy at RWTH Aachen University. It solves the coupled transient equations for groundwater flow, heat transport, and the transport of reactive solutes in porous media in three space dimensions. The program is written in Fortran and parallelized with OpenMP on two levels. The outer parallel level computes different directional derivatives needed for the computation. In the inner level for each derivative a linear system of equations is assembled and solved in parallel. We used a big dataset computing up to 16 directional derivatives, thus limiting the outer parallel level to 16 threads.

We run tests with and without using our binding library, using a `scatter` strategy on the outer level and a `subscatter` strategy on the inner one. The performance results for the Tigerton machine with different numbers of inner and outer threads is given in Fig. 4. The results for the bound and the unbound case do not differ very much.

When 1x4, 1x8 and 2x4 threads are used the unbound case is better than the bound case. The reason for this behavior is the strategy we use. When the threads are placed by our library one socket for the 1x4 case and two sockets for the 1x8 and 2x4 cases are used. In the unbound case the threads can run on all four sockets, which leads to a slightly better performance. We could change the strategy for these cases, but then we would have to use different strategies for the different thread numbers. This would oppose our aims for ease of use and performance portability. Furthermore, the cases which are most interesting for the user are of course the cases with the best performance. Starting 1x4 threads and leaving the rest of the machine empty does not make much sense when a better performance could be achieved with 1x16 threads. For the cases where the whole machine is used, the differences between the bound and unbound cases are marginal.

The results for the Barcelona machine are shown in Fig. 5 and display a different behavior. Large differences for smaller thread numbers are not observed, even the runs with 1x4, 1x8 and 2x4 threads do not show significant differences in the speedup, as they did on the Tigerton machine. On the Tigerton machine, the application profits in the unbound case from using four sockets whereas just two sockets are used in the bound case. Here we use the same strategy, so again we use two sockets with our library instead

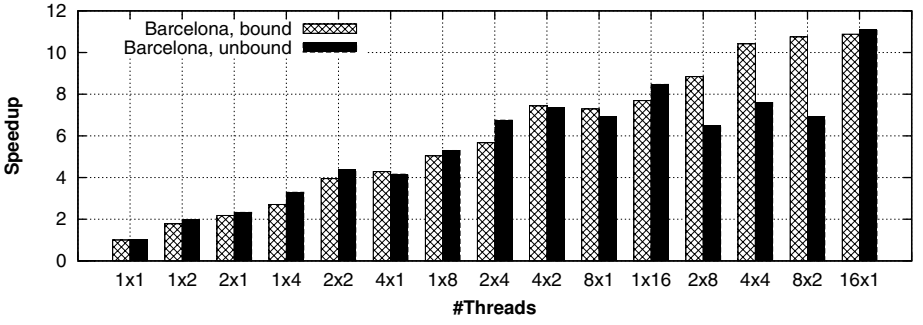


Fig. 5. Best speedup for the SHEMAT-Suite code on the Barcelona machine

of four sockets without it. But most of the data is initialized before the inner regions are opened. On the Barcelona machine the data is initialized in the local memory of one processor in the 1x4 and 1x8 cases. In the 2x4 case we observed Linux placing the outer threads on one or two processors. Therefore, we do not profit from using four sockets in the unbound case. The data is accessed and loaded from remote memory. This does not improve when using four sockets. On the Tigerton machine the initialization is not important, because it is an SMP machine, achieving a higher total memory bandwidth when all sockets are used. On the other hand, for 16 threads we see, on the Barcelona machine, that the placement of threads is important. For the cases where just one parallel level is used, i.e. 1x16 and 16x1, the difference is not as high as for the other cases. We observe an advantage of 50% - 70% in the nested cases 2x8, 4x4 and 8x2. Even for the case with the largest speedup (16x1), we can see a benefit of about 6%.

As described in Section 1, the binding problems are much more important for nested OpenMP. When we use just one thread on the outer level or one thread on the inner level, the program behaves nearly as if there would be no other level, so we do not profit from our binding strategies significantly.

We also perform experiments on the ScaleMP machine; the results are shown in Fig. 6. Here, we see a large difference between the cases with and without binding. For small numbers of threads, the behavior is nearly the same. This is because the first threads can run on the same board and then we have the same behavior as on an SMP system. But with a growing number of threads, binding really pays off. Without binding the best speedup is less than 8.5 and with the use of our library we achieve a speedup of 20.33 for 10x8 threads. So, the performance improvement is more than 140%. Another interesting point is that the total speedup is above 20 which is much more than we obtain on the other machines. In summary, the SHEMAT-Suite code scales better on the ScaleMP machine than on the other two machines, provided that an appropriate binding is used.

7 Conclusion

Looking at benchmark kernels and real-world applications we demonstrate that the placement of threads for nested OpenMP programs is an important issue for perfor-

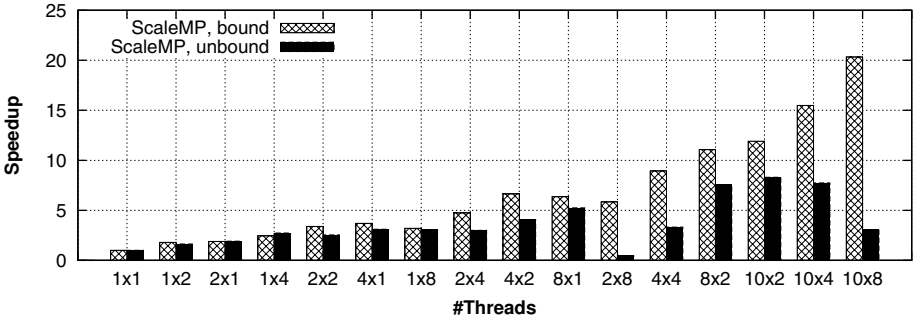


Fig. 6. Best speedup for the SHEMAT-Suite code on the ScaleMP machine

mance. On systems with a flat memory (traditional SMP), like the Tigerton machine, it turns out that thread placement is of less importance for the application codes. But on cc-NUMA machine (like Barcelona), we improve the performance with little extra work for the programmer by employing our novel binding library.

Given that all major system vendors currently build cc-NUMA architectures, we stress that it will be increasingly important to take care of these characteristics in the near future.

For the ScaleMP machine it is even more important to place the threads in a suitable way because the remote accesses are more expensive than on the Barcelona machine.

Our experiments reveal that a particular real-world application indeed profits from the characteristics of this special machine. Furthermore, the results gathered on this machine are of special interest as they clearly indicate that with a growing number of cores and memory hierarchies getting deeper the optimization opportunities of multi-level parallel programs are not sufficiently supported by the current OpenMP 3.0 standard.

Our proposal based on thread binding strategies for each parallelization level or even for each parallel region is easy to use and still exploits the performance potential of these machines.

Acknowledgments

The authors would like to thank our colleagues at the Institute of Aerodynamics and at the Institute for Applied Geophysics for making available their computer codes.

References

1. Ayguad, E., Martorell, X., Labarta, J., Gonzlez, M., Navarro, N.: Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In: Proc. of the 1999 International Conference on Parallel Processing, Ajzu, pp. 172–180 (1999)
2. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
3. Clauser, C. (ed.): Shemat and Processing Shemat - Numerical simulation of reactive flow in hot aquifers. Springer, Berlin (2002)

4. Hörschler, I., Meinke, M., Schröder, W.: Numerical simulation of the flow field in a model of the nasal cavity. Computers & Fluids 32(1), 39–45 (2003)
5. Johnson, S., Leggett, P., Ierotheou, C., Spiegel, A., an Mey, D., Hörschler, I.: Nested Parallelization of the Flow Solver TFS using the ParaWise Parallelization Environment. 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. 217–229. Springer, Heidelberg (2008)
6. Huang, L., Chapman, B., Liao, C.: An Implementation and Evaluation of Thread Subteam for OpenMP Extensions. In: Workshop on Programming Models for Ubiquitous Parallelism (PMUP 06), Seattle (2006)
7. 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)
8. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and Prototype of a Performance Tool Interface for OpenMP. J. Supercomput. 23(1), 105–128 (2002)
9. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: MAW '08: Proceedings of the 2008 workshop on memory access on future processors, pp. 377–384. ACM, New York (2008)
10. 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)
11. Zhang, G.: Extending the OpenMP Standard for Thread Mapping and Grouping. 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. 435–446. Springer, Heidelberg (2008)

A Proposal for User-Defined Reductions in OpenMP

Alejandro Duran[1], Roger Ferrer[1], Michael Klemm[2],
Bronis R. de Supinski[3], and Eduard Ayguadé[1,4]

[1] Barcelona Supercomputing Center
{alex.duran,roger.ferrer,eduard}@bsc.es
[2] Intel Corporation
michael.klemm@intel.com
[3] Lawrence Livermore National Laboratory
bronis@llnl.gov
[4] Universitat Politècnica de Catalunya

Abstract. Reductions are commonly used in parallel programs to produce a global result from partial results computed in parallel. Currently, OpenMP only supports reductions for primitive data types and a limited set of base language operators. This is a significant limitation for those applications that employ user-defined data types (e. g., objects). Implementing manual reduction algorithms makes software development more complex and error-prone. Additionally, an OpenMP runtime system cannot optimize a manual reduction algorithm in ways typically applied to reductions on primitive types. In this paper, we propose new mechanisms to allow the use of most pre-existing binary functions on user-defined data types as User-Defined Reduction (UDR) operators. Our measurements show that our UDR prototype implementation provides consistently good performance across a range of thread counts without increasing general runtime overheads.

1 Introduction

OpenMP [19] is a well-known and widespread programming model for the development of parallel applications on shared-memory platforms. It allows parallel and sequential implementations to co-exist in a single code base by using directives that tell the compiler which parts of the code to parallelize. Non-OpenMP compilers safely ignore the parallelization hints and emit an executable for sequential execution. In practice, OpenMP does not always keep its single-source promise since programmers often must modify their sequential code to overcome OpenMP's limitations.

We focus on OpenMP's lack of support for arbitrary reduction operators on arbitrary data types in this paper. Programmers use reductions to produce a global result from partial results computed in parallel. OpenMP currently only supports reductions for primitive data types and a limited set of base language operators. If the program computes a reduction on a user-defined data type or with a more complex operator, the programmer must implement the reduction algorithm manually. This limitation makes errors likely and complicates program maintenance by requiring repeated implementation of a common design pattern. Performance may also suffer since the OpenMP

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 43–55, 2010.

implementation can no longer adapt the reduction algorithm to the specific aspects of the execution (e. g., thread count or architecture).

In this paper, we propose extensions to OpenMP that eliminate this limitation. Our solution provides an additional OpenMP declarative directive `declare reduction` that specifies that a binary function (a *UDR operator*) can be used as a *User-Defined Reduction* (UDR). We also extend OpenMP's `reduction` clause to accept UDR operators in addition to built-in OpenMP operators. While our implementation is as efficient as highly optimized manual idioms for user-defined reductions, it provides a more concise and easy-to-use syntax to express user-defined reductions.

The remainder of the paper is organized as follows. The following section discusses UDR mechanisms in other parallel programming languages. We then elaborate on the limitations of the current OpenMP specification in Section 3. Section 4 details our proposal for UDR support in OpenMP. Finally, we evaluate performance of our proposal in Section 5. Overall, we show that our prototype implementation outperforms many hand-coded UDRs, particularly with large thread counts, without implying other overheads.

2 Related Work

Parallel programming frequently requires aggregation of local (partial) results into a global result. While low-level threading APIs such as POSIX threads [9], Windows Threads [16], Java Threads [18], or C# Threads [15] allow programmers to implement UDRs manually, other parallel programming languages provide a better, higher level approach.

Google's MapReduce API [4] provides a parallelization API that distributes work and accepts user-supplied reducers. Although MapReduce supports object-oriented languages, it only processes key-value data. Our OpenMP UDR proposal reflects the philosophy of OpenMP by using compiler directives to define UDRs, which supports all OpenMP base languages. Further, we directly support the variety of data types available in those languages.

MPI [17] includes UDR support. While our approach accepts a wide range of binary functions, MPI UDRs are restricted to a special function signature. The programmer must provide corresponding wrapper functions to reuse existing functions. Additionally, we use a declarative syntax to define UDRs, whereas MPI requires special allocate and free function calls to inform the runtime about UDRs.

ZPL [5] relies on overloading for the specification of associative and commutative UDRs. One signature of the function returns the identity element while a second signature implements the actual reduction operator. OpenMP/Java [13] extends OpenMP with a `Reducer` interface to define UDRs. Similarly to ZPL, the interface requires separate methods to return the identity and to reduce two values. Cilk++ [7] follows a similar approach that defines special classes that implement the reduction semantics. Unlike these UDR approaches, we use explicit clauses of the UDR declaration to specify the identity element and the reduction operator and, thus, support all OpenMP base languages.

TBB [20] parallelizes C++ programs through templates to which reductions can be added as methods. PPL [14] also supports UDRs through `combinable` objects. While

```
1 void example(double *array, size_t N) {
2     double sum = 0.0;
3     double prd = 1.0;
4 #pragma omp parallel for reduction(+:sum) reduction(*:prd)
5     for (size_t i = 0; i < N; i++) {
6         sum += array[i];
7         prd *= array[i];
8     }
9 }
```

Fig. 1. Simple reduction example with two reduction variables

TBB and PPL provide UDRs, they only cover C++ programs; our approach is more generic in that it also targets C and Fortran. Our declaration syntax also provides better separation of concerns as a UDR can be reused in any parallel region, whereas TBB and PPL programmers must re-implement the reduction method in all functors, effectively adding redundant code to the application.

Kambadur, et al. previously proposed a mechanism to support user-defined reductions in OpenMP [12]. Their proposal targets C++0x [6] programs and makes heavy use of C++ concepts [8] outside of OpenMP pragmas. While this approach effectively solves the problem of adding user-defined reductions to C++0x, our approach is more generic as it extends also to C and Fortran. In addition, we rely on a pragma-only syntax to supply all required information to employ user-defined reductions in OpenMP applications.

3 Costs of the Lack of UDR Support in OpenMP 3.0

OpenMP 3.0 only provides a standard set of reduction operators (e. g., addition) that operate on built-in primitive data types (e. g., `double`) of the corresponding base language. Fig. 1 shows an example of an OpenMP `parallel` construct (line 4) that performs simple sum and product reductions on the variables `sum` and `prd`.

In the example, the variables are of the primitive data type `double`. If we change them to a user-defined data type such as `complex_t`[1] we can no longer use OpenMP reductions. Instead, we must manually implement the reduction algorithms using one of the many ways to write a parallel reduction. Fig. 2 presents an efficient reduction algorithm that has little overhead for small thread counts.

The code in Fig. 2 first determines an upper bound of the possible number of participating threads (line 9). We then declare temporary arrays for each reduction variable to hold the intermediate local results of each thread (lines 11–12).[2]

Each thread has a private copy of the reduction variables `sum` and `prd`. Since these private copies are in different stacks, they cannot cause false sharing. Each thread must initialize its copies with the appropriate identity values (lines 16 and 17). Each thread then computes a local reduction for the array elements corresponding to its iterations

[1] We use this type for explanatory purposes despite the `_Complex` primitive type in C99.

[2] For simplicity, the code shown could allocate more space than necessary since some OpenMP threads may not participate in the `parallel` region.

```
1 complex_t complex_add(complex_t a, complex_t b);
2 complex_t complex_mul(complex_t a, complex_t b);
3
4 void example(complex_t *array, size_t N) {
5     int nthreads;
6     complex_t sum = {0.0, 0.0}, prd = {1.0, 0.0};
7     complex_t *part_sum, *part_prd;
8
9     nthreads = omp_get_max_threads();
10
11    complex_t part_sum[nthreads];
12    complex_t part_prd[nthreads];
13
14 #pragma omp parallel shared(part_sum,part_prd) private(sum,prd)
15    {
16        sum = {0.0, 0.0};
17        prd = {1.0, 0.0};
18 #pragma omp for
19        for (size_t i = 0; i < N; i++) {
20            sum = complex_add(sum, array[i]);
21            prd = complex_mul(prd, array[i]);
22        }
23        part_sum[omp_get_thread_num()] = sum;
24        part_prd[omp_get_thread_num()] = prd;
25    }
26    for (int thr = 0; thr < nthreads; thr++) {
27        sum = complex_add(sum, part_sum[thr]);
28        prd = complex_mul(prd, part_prd[thr]);
29    }
30 }
```

Fig. 2. Programming pattern for user-defined reductions in OpenMP 3.0

of the `for` loop (lines 18–22). After executing the loop, each thread stores its partial results in the temporary arrays (line 23 and 24). After the parallel region, the master thread iterates over all partial results in the temporary arrays and produces the final result of the computation (lines 26–29).

Although this implementation performs well for small thread counts, larger thread counts might benefit from a tree-based reduction. Tree-based reductions are significantly more complicated and require a much higher coding effort. Switching between these implementations would require even more complex code, which must be repeated for every UDR in the OpenMP application. Although the pattern could be provided by a parametrized library function, direct OpenMP support for UDRs would be less error-prone and more efficient.

Fig. 3 shows how our proposal simplifies this example. In lines 4 and 5, declaration pragmas inform the OpenMP compiler about UDR operators on the type `complex_t` and supply the corresponding identity values. After definition, the `reduction` clause can use these UDR operators (line 9), resulting in code almost identical to that for the `double` primitive type of Fig. 1.

4 User-Defined Reductions for OpenMP

This section explores the design space for user-defined reductions and presents our `declare reduction` directive and the modifications to the current `reduction`

```
1 complex_t complex_add(complex_t a, complex_t b);
2 complex_t complex_mul(complex_t a, complex_t b);
3
4 #pragma omp declare reduction(complex_add:complex_t) identity({0.0,0.0})
5 #pragma omp declare reduction(complex_mul:complex_t) identity({1.0,0.0})
6
7 void example(complex_t *array, size_t N) {
8     complex_t sum = {0.0, 0.0}, prd = {1.0, 0.0};
9 #pragma omp parallel for reduction(complex_add:sum) reduction(complex_mul:prd)
10    for (size_t i = 0; i < N; i++) {
11        sum = complex_add(sum, array[i]);
12        prd = complex_mul(prd, array[i]);
13    }
14 }
```

Fig. 3. Example of Fig. 2 rewritten with user-defined reductions

clause. We then discuss extensions that support UDRs on array types and that more tightly integrate them with object-oriented languages.

4.1 Design Rationale

The UDR language extension is subject to several crucial design requirements. First, it must follow the OpenMP directive-based philosophy. Second, the UDR feature must blend well with all OpenMP base languages and reflect their specifics while maintaining a common syntax across them. Third, the mechanism must express UDRs without any unnecessary syntax bloat. Fourth, the definition should allow for efficient implementations when used with any parallel loop schedule and support common optimizations.

The OpenMP compiler needs two pieces of information to implement a reduction: the identity element and the implementation of the operator. It needs the operator's identity value to initialize temporary variables that hold intermediate results. The implementation must combine two input values into one output value. The compiler generates code that invokes the reduction operator whenever the reduction algorithm aggregates values from different threads. The OpenMP specification provides this information for the reductions supported in OpenMP 3.0, while programmers must supply it for UDRs.

We could simply extend the existing `reduction` clause, which would not add new any idioms to OpenMP. However, programmers would have to supply the above information at every `reduction` clause that works on a user-defined data type. This unnecessary repetition would increase the likelihood of errors at the `reduction` clauses.

Thus, we split the UDR definition into two parts: UDR *declaration* and UDR *usage*. At the declaration, programmers describe UDRs by specifying the UDR operator and the identity value. OpenMP-enabled libraries can safely incorporate UDR declarations for their data types in C or C++ header files or Fortran modules. At UDR usage, programmers supply the declared UDR name in a `reduction` clause as the operator.

In contrast to designs for UDRs in other programming models, one of our main design principles is code reuse. We explicitly allow programmers to reuse existing binary functions without the need for any wrapper mechanisms that adapt existing code interfaces to UDR requirements. Most OpenMP programs stem from a sequential code base with a set of operators on user-defined data types. These operators often include

functions with two input values and one output value and that UDRs can reuse. Thus, our mechanism blends well with OpenMP's principle of incremental parallelization.

A sequential loop, such as that corresponding to the example in Fig. 1, combines array elements in the sequential iteration order. If the OpenMP implementation assigns a single chunk of iterations to each thread, then the reductions performed within each thread will be subsequences of the sequential iteration order. If the compiler combines these temporary values in the chunk order, the overall reduction order is simply a reassociation of the original computation. Thus, we require UDR operator to be *associative* so that the implementation can compute the reduction using multiple threads without sequentializing (e. g., using a critical region).

OpenMP schedules allow different distributions of iterations to threads besides a single chunk of consecutive iterations. These schedules reorder the operations. The original OpenMP intrinsic reduction operators are all associative and *commutative*,[3] which allows the implementation to combine iterations into a partial reduction in each thread and then combine them into the global result in any order, such as the order in which the threads complete. While we could restrict the loop schedules or require the use of the `ordered` clause, we require the UDR operators to be commutative in order to maximize parallelism and to simplify the implementation of UDRs.

4.2 The `Declare Reduction` Directive

UDRs must be declared prior to their use in a `reduction` clause. We use the `declare reduction` directive with the following syntax:[4]

```
1 #pragma omp declare reduction(op-list:type-list) [clause]
```

where *op-list* is a comma-separated list of UDR operators, and *type-list* is a comma-separated list of defined data types. Clause can only be an `identity` clause.

Basic Syntax. The `declare reduction` directive instructs the compiler that the *operators* in the list are valid *UDR operators* for the types specified in the *type-list*. The directive can be specified for any type (primitive types and user-defined data types) except functions and array types. Reductions on function types do not have any useful semantics; arrays are handled differently (see Section 4.4).

A valid UDR operator *op* must exist for each type. As we strive to maximize code reuse we define a set of minimum requirements for a possible UDR operator to be valid instead of defining a fixed prototype to which all operators must conform. These requirements are the following:

- *op* must be a *binary* function with both arguments of a type compatible with the type in the UDR declaration;[5]
- *op* must be a commutative function;
- *op* must be an associative function;

[3] Although the subtraction operator is non-commutative, it is mapped to the commutative addition operator by many OpenMP implementations.

[4] We only present the C/C++ syntax and requirements, which are similar to those of Fortran.

[5] With *T* being the type in the UDR declaration, compatible types in C or C++ include *T*, *const T*, *T* *, *T* &, *const T* * and *const T* &.

```
1 struct T {
2     void alpha ( const T & );
3     const T & operator+ ( const T & );
4 };
5
6 T& alpha ( T &, T & );
7 T  beta  ( T *, T );
8 void  gamma ( T *, T );
9 void  delta ( T *, T *);
10 void  epsilon ( T, T *);
11
12 const T & operator* ( const T & , const T & );
13
14 #pragma omp declare reduction(+,*,T::alpha,alpha,beta,gamma,delta : T)
```

Fig. 4. C++ examples of valid UDR operators for data type *T*

- *op* must produce a result in its function return value or an argument; if *op* could produce multiple results (e. g., both arguments are pointer types) then the leftmost result is used (i. e., the precedence is return value, left argument, right argument).

For large data structures, it might prove inefficient to allocate a temporary copy of a data structure to store the (partial) reduction result and then pass the temporary as the return value of the UDR operator. Hence, we allow the UDR operator to place the (partial) reduction result in one of its arguments to avoid these unnecessary copies of large data structures. While this choice does not incur higher implementation overhead in the OpenMP compiler, it provides additional flexibility to programmers to optimize their user-defined reductions.

For C++, all standard and (correctly implemented) overloaded operators are valid UDR operators; function members of that type are valid if they have a single argument compatible with the type. In any case, the function must be accessible and unambiguous in the scope where the reduction takes place as well as in the scope where the UDR is declared.

Fig. 3 provided valid UDR declarations for our simple C example used in Section 3. Fig. 4 provides additional examples of valid UDR declarations in C++ for a user-defined data type *T*. The operators + and * are overloaded C++ operators for *T* that are visible in the scope that contains the UDR declaration. The UDR operator *T::alpha* refers to the member function of *T*; *alpha* refers to the global function. The functions *gamma* and *delta* are valid UDR operators since they take two input values of type *T*. Both *gamma* and *delta* must store the reduction result in the left argument since they do not have return values. Similarly, *epsilon* must store it in the right argument.

The `identity` clause. By default, we perform *zero initialization* for *non-object types* and invoke the default constructor for *object types* in C++. The `identity` clause overrides the default with user-defined values. It takes either a constant expression, a brace initializer, or the special keyword `constructor` and a list of constant expressions of the form *(expr1,...,exprN)*. In the first two cases, all temporaries are assigned the identity value initially. In the last case, the constructor for the specified type is invoked with the listed arguments. Fig. 5 shows examples of these different cases.

```
1 #pragma omp declare reduction(fixed_mul:fixed_t) identity(1)
2 #pragma omp declare reduction(complex_mul:complex_t) identity({1.0,0.0})
3 #pragma omp declare reduction(*:Complex) identity(constructor(1.0,0.0))
```

Fig. 5. Examples of valid `identity` clauses

```
1 #pragma omp declare reduction(matrix_add:int[][])
2
3 void example() {
4    int M[n][n];
5
6 #pragma omp for reduction(matrix_add:M)
7    for(...) {...}
8 }
```

Fig. 6. Example of a UDR on a two-dimensional array

4.3 Extensions to the `Reduction` Clause

Our UDR proposal does not change the well-known syntax of the `reduction` clause. We only require that it accepts declared UDR operators as well as the built-in reduction operators (and intrinsic functions in Fortran). When a UDR operator is specified in a `reduction` clause, the OpenMP compiler must determine the UDR declaration that applies to the scope of that particular `reduction` clause. It then uses the information from the UDR declaration to implement the reduction. First, the compiler initializes any temporaries with the identity value. Second, it replaces occurrences of the original variable with the corresponding private temporary variable. Finally, it uses the UDR operator in its reduction algorithm to combine the temporaries into the overall result. As all necessary information is specified at the UDR declaration, the compiler can implement more sophisticated reduction approaches as well.

4.4 Array Reductions

OpenMP 3.0 supports array reductions on primitive types for Fortran but not for C or C++ because the number of dimensions (and their size) may not be available to the compiler at the `reduction` clause in those languages.

If the reduction variable is strictly an array, the compiler could infer the number of dimensions and the size from the reduction variable. But, we require the programmer to add square brackets to the data type in the UDR *declaration* to specify that the operator will work for array types. This allows the compiler to check at the UDR *declaration* that the operator is valid for the type. The actual size of the dimensions, which are needed to create the correct private variables, is deduced by the compiler from the type of the variable of the `reduction` clause.

Fig. 6 declares a UDR on a two-dimensional matrix of `int` values by specifying `int[][]` as the UDR's data type. A compiler allocates private arrays of $n \times n$ elements (see line 4) and initializes each element with the identity value (i. e., in this case as no `identity` clause is specified, with the value 0) for the UDR usage in line 6.

```
1 #pragma omp declare reduction(vector_add:int[])
2
3 void example(int *a, int n) {
4   int (*v)[n] = (int (*)[n]) a;
5
6 #pragma omp for reduction(vector_add:v)
7   for(...) {...}
8 }
```

Fig. 7. Example of UDR for arrays with pointer reshaping

```
2 #pragma omp declare reduction(template <typename T_> + : std::vector<T_>)
```

Fig. 8. UDR for adding `std::vector` objects

While this works well for variables that are strictly arrays, an issue in C and C++ is that arrays are implicitly converted to pointers across call boundaries [10,11]. To improve the support of UDRs for arrays, our proposal considers *pointers to arrays* as if they were arrays for the purpose of finding the corresponding UDR. As Fig. 7 shows, programmers must often convert *pointer to types* to *pointers to arrays* that contain the needed dimensionality information. Although this solution requires some modifications to the sequential code, the sequential code remains valid and we avoid more extensive shaping expression support for the UDR.

4.5 C++-Specific Extensions

Although our UDR design provides a common syntax for C, C++, and Fortran, we extend the UDR syntax to make UDR declarations more concise and easier to use in C++. The first extension targets C++ templates (i. e., partially instantiated types):

```
1 #pragma omp declare reduction(template<template-header> op-list:type-list)[clause]
```

After specifying a *template-header*, the different template parameters defined in the header may appear in the *operator-list*, *type-list*, or the `identity` clause.

Template support is crucial for C++ to define UDRs for template types in a generic way. For instance, the template `std::vector<T_>` of Fig. 8 defines reductions on any possible vector. Otherwise, possible instantiations (e. g., `std::vector<int>`, `std::vector<float>`) would require separate UDR declarations.

In addition to template support, we use the *dot syntax* to omit class qualifiers in both the `declare reduction` directive and the `reduction` clause. Fig. 9 shows how it simplifies the use of qualified C++ identifiers in UDRs. Qualifiers for identifiers can be omitted if a dot prefixes UDR operator names. The compiler then automatically qualifies the operator with the type(s) being declared (in UDR declaration directives) or of variables of `reduction` clauses. This syntax can greatly simplify the declaration and usage of UDR operators that have the same *name* on unrelated types but that implement the same kind of reduction.

```
1 namespace A { class T; }
2 namespace B { class S; }
3
4 // declares UDRs A::T::foo, B::S::foo, A::T::bar, and B::S::bar
5 // with the same identity
6 #pragma omp declare reduction(.foo,.bar: A::T,B::S)
7
8 ...
9
10 A::T t;
11 B::S s;
12
13 // Uses UDR A::T::foo for t and B::S::foo for s
14 #pragma omp parallel for reduction(.foo:t,s)
15 ...
```

Fig. 9. Example of the dot syntax to shorten UDR declarations of qualified identifiers

5 Evaluation

Although our proposed extensions to OpenMP simplify reduction operations for non-basic types it remains to be seen if they can be implemented as efficiently as hand-made reductions.

To this purpose we implemented a prototype of our proposal for user-defined reductions in the Mercurium source-to-source compiler [2]. The code generated for standard OpenMP reductions and UDRs is the same except that it uses the identity and operator that the UDR declaration specifies.

We have implemented five kinds of reductions that capture typical OpenMP reduction scenarios:

- Our *standard reduction* tests the existing reduction support with an `int` sum;
- Our *manual critical* version stores partial values of each thread in a temporary variable, which it sums into a shared variable in a `critical` region at the end of the parallel region;
- Our *manual atomic* variant is identical to our *manual critical* version except that it performs the sum in an `atomic` construct, which generic UDRs cannot use—we evaluated this reduction strategy for completeness;
- Our *manual shared arrays* test uses the reduction algorithm that Fig. 2 shows;
- Our *UDR* version uses a UDR operator that adds two `int` variables and returns the result as the return value.

We use the statistical analysis of the EPCC OpenMP benchmark suite [1] but determine the overhead of reductions directly. Specifically, we profile the time that each thread spends in the reduction operation rather than indirectly measuring the overhead by subtracting the overhead of a `parallel` region from the overhead of a `parallel` region with a `reduction` clause. The indirect method has high variance since the reduction overhead is relatively small compared to that of any parallel region.

We measure reduction execution times with the Itanium interval timer facilities [3] of an SGI Altix 4700 (1.67 GHz). We vary the thread count from one to 64 and compute the average of 2,000 overhead measurements for each kind of reduction. The maximum

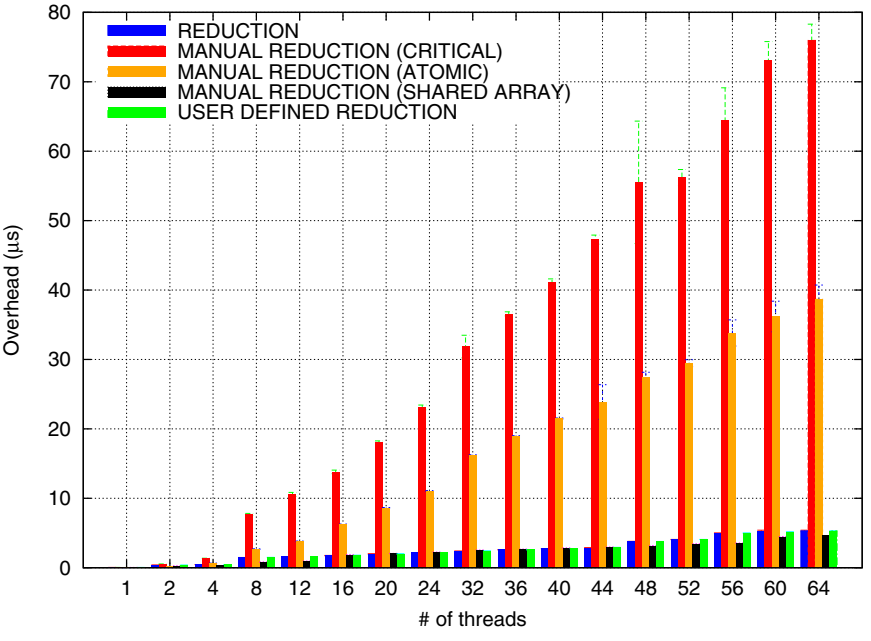


Fig. 10. Performance comparison of reduction patterns and UDRs

deviation of the time measured is around 2%, which indicates that our methodology provides a consistent measurement. Fig. 10 summarizes our microbenchmark results.

The *manual critical* test incurs higher reduction overheads with increasing thread counts. As the thread count increases, more threads compete for the central lock that protects the computation of the global result. With two threads, the overhead to enter and to exit the `critical` region already exceeds the time spent within it. As the threads reach the `critical` region at about the same time, the overhead increases linearly with the thread count.

The *manual atomic* variant improves performance over the *manual critical* version by roughly 50% but we still observe linear increases in overhead with thread count. The *manual atomic* variant incurs smaller overhead since `atomic` uses atomic instructions instead of acquiring and releasing a lock. The machine's memory subsystem ensures mutual exclusion for accesses to the global result, which still incurs increasing overhead as the thread count increases. Each step of the reduction also incurs a cache fault.

The *standard reduction*, *manual shared array*, and *UDR* tests exhibit roughly the same overhead. Mercurium implements standard OpenMP reductions with private temporary variables on each threads' stack. A `for` loop then retrieves each thread's private variable and adds them to the global result at the end of the `parallel` region. Our *manual shared arrays* implementation uses the same approach except that it stores the private variables in a shared array. Our UDR implementation extends Mercurium's standard reduction algorithm such that it invokes the UDR function when computing

the global result. The native compiler inlines the UDR function so it incurs almost no additional overhead compared to the *standard reduction* and *manual shared array* tests.

In summary, our performance evaluation shows that the UDR implementation of our prototype exhibits the same level of performance as standard reductions and the efficient manual UDR implementation of Fig. 2. Thus, we demonstrate that OpenMP can remove the burden of error-prone and cumbersome manual idioms for reductions of user-defined types from the programmer while providing high performance through our UDR mechanism.

6 Conclusions and Future Work

OpenMP applications, like their sequential counterparts, often employ user-defined data types. Typically, programmers must overcome OpenMP's lack of support for reductions on these types. Our new mechanism overcomes this limitation by concisely specifying user-defined reductions in OpenMP programs. Our solution uses a declarative directive that is consistent with existing OpenMP syntax and allows existing binary functions on user-defined data types to serve as UDRs. UDRs support all OpenMP base languages and blend well with potential future OpenMP base languages.

Our UDR mechanism allows the OpenMP runtime system to choose the most efficient reduction algorithm for a parallel region. For example, the runtime can adapt the reduction algorithm to the thread count, which would otherwise require complex user programming. Our measurements have shown that our proposal introduces no additional overhead compared to manually implemented reductions (or regular OpenMP reductions) while avoiding copy-and-paste duplication of reduction algorithms and hard-to-find errors that stem from user-level reduction implementations.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC Network of Excellence (contract FP7/ICT 217068), and the MareIncognito project under the BSC-IBM collaboration agreement.

References

1. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proc. of 1st European Workshop on OpenMP, Lund, Sweden, October 1999, pp. 99–105 (1999)
2. Barcelona Supercomputing Center. The NANOS Group Site: The Mercurium Compiler, `http://nanos.ac.upc.edu/mcxx`
3. Intel Corporation. Intel Itanium 2 Processor Reference Manual for Software Development and Optimization (May 2004); Order number 251110-003
4. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of ACM 51(1), 107–113 (2008)

5. Deitz, S.J., Chamberlain, B.L., Snyder, L.: High-level Language Support for User-defined Reductions. Journal of Supercomputing 23(1), 23–37 (2002)
6. Becker, P. (ed.): Working Draft: Standard for Programming Language C++ (November 2009); Document number N3000
7. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and Other Cilk++ Hyperobjects. In: Proc. of the 21st Ann. Symp. on Parallelism in Algorithms and Architectures, Calgary, AB, Canada, August 2009, pp. 79–90 (2009)
8. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Lumsdaine, A., Dos Reis, G.: Concepts: Linguistic Support for Generic Programming in C++. In: Proc. of the 2006 ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, Portland, OR, October 2006, pp. 291–310 (2006)
9. IEEE. Threads Extension for Portable Operating Systems (Draft 6), Document P1003.4a/D6 (February 1992)
10. ISO/IEC. Programming Languages – C, ISO/IEC 9899:1999 (1999)
11. ISO/IEC. Programming Languages – C++, ISO/IEC 14882:2003 (2003)
12. 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)
13. Klemm, M., Veldema, R., Bezold, M., Philippsen, M.: Proposal for OpenMP for Java. 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. 409–421. Springer, Heidelberg (2008)
14. McGrady, D.: Avoiding Contention using Combinable Objects (September 2008), `http://blogs.msdn.com/nativeconcurrency/archive/2008/09/25/avoiding-contention-using-combinable-objects.aspx`
15. Michaelis, M.: Essential C# 3.0: For.NET Framework 3.5 (Microsoft.Net Development), 2nd edn. Addison-Wesley Longman, Amsterdam (September 2008)
16. Microsoft Developer Network. Process and Thread Functions (Windows), `http://msdn.microsoft.com/en-us/library/ms684847%28VS.85%29aspx`
17. MPI Forum. MPI: Extensions to the Message-passing Interface, Version 2.2. Technical report, MPI Forum (September 2009)
18. Oaks, S., Wong, H.: Java Threads, 3rd edn. O'Reilly, Sebastopol (2004)
19. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
20. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Sebastopol (July 2007)

An Extension to Improve OpenMP Tasking Control

Eduard Ayguadé[1], James Beyer[4], Alejandro Duran[1], Roger Ferrer[1], Grant Haab[5], Kelvin Li[3], and Federico Massaioli[2]

[1] Barcelona Supercomputing Center
{alex.duran,roger.ferrer,eduard}@bsc.es
[2] CASPUR.
federico.massaioli@caspur.it
[3] IBM Canada Lab
kli@ca.ibm.com
[4] Cray Inc.
beyerj@cray.com
[5] Intel Corp.
grant.haab@intel.com

Abstract. OpenMP tasks were introduced in order to support irregular parallelism. However, task runtime overhead is necessarily higher than for worksharing constructs, and can hamper performance if the tasks are too finely grained. In this paper, we address the common use case, of tasks generated in a tree-like hierarchy, with task granularity decreasing with increasing depth, and propose a new **`final`** clause to force coalescing of excessively fine grained tasks.

1 Introduction

The OpenMP *Tasking Model* is the most significant addition to OpenMP API V3.0 [7], allowing irregular parallelism to be expressed in an OpenMP program. As a consequence of the principles adopted during the design process [2], safety and correctness was always preferred over performance. Moreover, many possibilities explored during the design phase did not find their way into the 3.0 API. In particular, all features not supported by clear use cases or experimental evidence from compiler developers or end users, no matter how promising they might be, remained on the drawing board. More than one year after OpenMP API 3.0 publication, sufficient experience has been collected from the field to warrant readdressing some features.

Regular parallelism, as in OpenMP worksharing constructs, allows for limiting execution overhead. In particular, for a loop construct: i) number of iterations is known on entrance, ii) a barrier will not be encountered before construct end, iii) static scheduling allows execution to be planned in advance, and iv) all scheduling options allow iterations to be scheduled in blocks.

Most interesting cases of irregular parallelism are not amenable to such optimizations. In general, the total number of generated tasks, number of children spawned by a task, and synchronization points cannot be known in advance. Hence, irregular parallelism incurs higher runtime overheads, which can dominate execution if the tasks are too fine grained. If task granularity is constant across an algorithm, it is quite easy

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 56–69, 2010.

for programmers to coalesce fine grained tasks together until overhead is acceptable. Unfortunately, coalescing is not that easy when tasks granularity changes dynamically, as is common in problems like adaptive configuration space exploration, or hierarchical grid or finite element discretization of mechanical systems [6], in which tasks are generated in an unbalanced tree-like hierarchy, and their granularity varies as a function of branch and depth. In this paper we show experimental analysis of this class of problems, and propose a **`final`** clause to control coalescing of child tasks.

2 Motivation

The **`task`** directive was added in the OpenMP API V3.0 [7] to support dynamic, irregular parallelism (e.g., while loops or recursive functions). A thread encountering a **`task`** construct generates a unit of work, the execution of which can be deferred with respect to the following code, or even handed to a different thread. For task-based application to scale, it is very important that developers control the overhead of task creation [1].

The **`task`** directive provides an optional **`if`** clause, that forbids deferral when its argument evaluates to *false*. Regardless of how the expression in the **`if`** clause evaluates, *true* or *false*, a new task data environment is always generated. When an **`if`** clause expression evaluates to *false*, the associated task is not subject to all scheduling possibilities, so some overhead could be avoided, at least in principle. Hence it would seem that the **`if`** clause is the solution in situations, like recursive algorithms, where computational cost shrinks as depth increases, and the benefit of generating a new task diminishes due to overhead cost. However, for safety reasons [2], variables referenced in a task construct are, in most cases, by default **`firstprivate`**, thus the overhead of data environment setup may be the dominant component of task creation. It could be thought that a clever and aggressively optimizing implementation could get rid of privatization, and coalesce the task altogether. Unfortunately, this is not generally possible, because variable privatization alters code semantics in very subtle ways, and proving that it is not needed can be difficult for a compiler. In other words, the semantics of the **`if`** clause do not allow effective control of task granularity. Moreover, as in the following example, Fig. 1, a task may require additional steps that are not needed when it is serialized. The only general approach is for users to avoid further task generation

```
1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2   ...
3   for (i = 0; i < n; i++)
4   #pragma omp task untied if(depth < MAX_DEPTH)
5   {
6     /* allocate a temporary array and copy <a> into it */
7     char * b = alloca((j + 1) * sizeof(char));
8     memcpy(b, a, j * sizeof(char));
9     b[j] = i;
10    if (ok(j + 1, b))
11      nqueens(n, j + 1, b, &csols[i], depth+1);
12  }
13  ...
14 }
```

Fig. 1. Avoiding fine granularity tasks with the **`if`** clause

manually by explicitly coding both parallel and serial versions of the algorithm, to be called under proper conditions.

The example in Fig. 1 shows the **if** clause specified on the **task** directive to avoid fine granularity tasks in the recursive routine `nqueens`. When the variable `depth` is larger than or equal to `MAX_DEPTH`, the granularity is too fine to benefit from task parallelism; it is more beneficial to execute the task immediately after it is generated. The problem is that there is overhead involved in complying with OpenMP semantics when the **if** clause evaluates to *false*.

Fig. 2 shows a manually modified version of the example in Fig. 1 in which the programmer provides a serial version of the `nqueens` routine (`nqueens_ser`). If the condition is satisfied, the serial code is executed. Hence, no task generation overhead is incurred. Furthermore, the serial routine is optimized with the knowledge that it will not spawn any new parallelism; in this particular case, the allocation and initialization of temporary array `b` has been removed.

```
1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2   ...
3   for (i = 0; i < n; i++)
4     if ( depth < MAX_DEPTH ) {
5       #pragma omp task untied
6       {
7         /* allocate a temporary array and copy <a> into it */
8         char * b = alloca((j + 1) * sizeof(char));
9         memcpy(b, a, j * sizeof(char));
10        b[j] = i;
11        if (ok(j + 1, b))
12          nqueens(n, j + 1, b,&csols[i],depth+1);
13      }
14    } else {
15      a[j] = i;
16      if (ok(j + 1, a))
17      nqueens_ser(n, j + 1, a,&csols[i]);
18    }
19  ...
20 }
```

Fig. 2. Avoiding fine granularity tasks manually

The performance benefit is shown in Fig. 3. The chart shows the speed-up for the version with the **if** clause, and two possible manual versions (one where the serial function has been optimized and one where it has not). The difference in performance between the version with the **if** clause and the unoptimized manual version is due to task generation overheads. While asking OpenMP to optimize the serial function is beyond today's compiler capabilities, we think that OpenMP should provide performance close to the unoptimized manual version.

The performance benefits above motivate the proposal of a new clause to the **task** directive – **final**. The **final** clause provides a way for users to control whether any child task should be generated and defines the appropriate semantics so it can be fully optimized by an OpenMP implementation. Additionally, we propose an additional API call that allows the user to specify parts of his code that do not need to be executed when a task is not spawned.

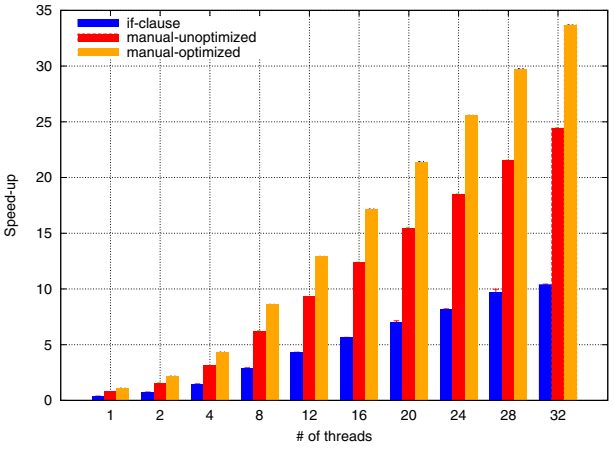


Fig. 3. Queens speed-ups controlling granularity

3 The Final Clause

In this section we propose the **`final`** clause with the appropriate semantics to efficiently control the granularity of tasks, overcoming the shortcomings of the existing **`if`** clause discussed in the previous section.

The syntax of this new clause is as follows:

```
1 final(expression)
```

When the **`final`** clause appears in a **`task`** construct the expression inside the clause is evaluated at task creation. If the clause evaluates to $true$ the task is considered a *leaf* or *final* task. All **`task`** constructs that are *closely nested* in the dynamic extent of a *final* task can be ignored by the implementation (either at compile or run time) and run in the context of the encountering task. Note, that if all **`task`** constructs are ignored then **`taskwait`** constructs can also be safely removed as they would effectively be *no-ops*.

There are two main differences between the **`if`** and the **`final`** clauses: the **`task`** constructs it affects and the way those constructs are "ignored".

While the **`if`** clause affects the task being defined by a **`task`** construct (and only that one), the **`final`** clause affects all its descendant tasks (even if they do not have a **`final`** clause themselves).

Compared to the **`if`** clause, where the **`task`** construct is only partially ignored (as we have discussed, a new task and data environment is still created when it evaluates to *false*), our proposed **`final`** clause semantics completely ignore the descendant **`task`** constructs (i.e., no new task nor new data environment will be created). OpenMP locks, **`critical`** and **`atomic`** constructs inside the dynamic extent of a *final* task still need to be honored to maintain the correctness of the program.

This difference in semantics allows the implementation to devise better optimizations to *inline* child tasks to increase the granularity in recursive patterns as the one shown

in Section 2. But, these semantics are a double-edged sword. The usage of the **`final`** clause can generate side-effects and it is not as safe as that of the **`if`** clause. These side-effects come because modifications of the data environment or *internal control variables*[7] would happen differently, depending if the task is *finalized* or not. Fig. 4 shows a case where the value of the variable `i` would change depending on the value of the expression `expr` inside the **`final`** clause.

This means the **`final`** clause is an optimization that users should only activate when they are sure it will not generate any problems (which is the common case because it follows the path of the serial execution).

```
1 void foo ()
2 {
3    int i = 3;
4    #pragma omp task final(expr) firstprivate(i)
5        i++;
6    printf("%d\n",i); // will print 3 or 4 depending on expr
7 }
```

Fig. 4. Example of side-effects due to the **`final`** clause

For completeness, we suggest that the **`final`** clause should also be allowed on the **`parallel`** and worksharing constructs. In this case, all **`task`** constructs in the region will be ignored. This is useful mainly for debugging purposes (i.e., disabling all tasks), but it could also be useful for performance reasons (e.g., in the presence of *nested parallelism*).

In addition we propose an API call, **`omp_in_final`**, that will return whether the current task is a *final task* or not. This call allows the programmer to check that a specific portion of his code is not called from a final clause if that could cause correctness problems. Also, it allows the programmer to optimize the code when task is final. For example, the example presented in Section 2 would look as Fig. 5 shows using the **`final`** clause and the **`omp_in_final`** call.

4 Implementation

The Mercurium C/C++ source-to-source compiler [4] was used to prototype the proposed **`final`** clause. Mercurium implements a subset of OpenMP 3.0 on top of the Nanos 4 runtime [8].

4.1 Implementation of Tasks

The Mercurium tasking implementation features a *creation mode* used when a **`task`** is about to be created. This creation mode, which is set at runtime, defines whether the task is actually created: if a task cannot be created it must run immediately. If no **`if`** clause is present the Nanos 4 runtime sets the creation mode thus enforcing a runtime task creation policy (e.g., to avoid overruns of runtime structures). Tasks with an **`if`**

```
1 void nqueens ( int n, int j, char *b, int *sol, int depth ) {
2   ...
3   for (i = 0; i < n; i++)
4   #pragma omp task untied final(depth+1 >= MAX_DEPTH)
5   {
6     char *b;
7     int  *sols;
8
9     if ( omp_in_final() && depth >= MAX_DEPTH ) {
10       b = a;
11       *sols = sol;
12    } else {
13       /* allocate a temporary array and copy <a> into it */
14       char * b = alloca((j + 1) * sizeof(char));
15       memcpy(b, a, j * sizeof(char));
16       *sols = &csols[i];
17    }
18
19    b[j] = i;
20    if (ok(j + 1, b))
21      nqueens(n, j + 1, b, sols, depth+1);
22  }
23  ...
24 }
```

Fig. 5. Avoiding fine granularity tasks with the **`if`** clause

clause whose expression evaluates to *false* have an *immediate* creation mode (effectively implementing the semantics of the **`if`** clause).

Mercurium implements tasks using the common technique of the *outline function* [3] where a new function is created containing the code of the task. The address of the outline function, along with the arguments, is passed to the runtime so the task can be enqueued for later execution.

Consider the listing in Fig. 6. The code generated by Mercurium to create the task inside function `f` (line 16) is shown in Fig. 7.

As we can see in Fig. 7, two paths can be followed depending on the creation mode. If the creation mode is *immediate* (this can happen because an **`if`** clause expression evaluates to *false* or because the runtime denies the creation of the task) the transformed code calls the outline function (line 23), effectively running the task code in the current thread. If the creation mode allows the creation of the task, then a task is created by means of a runtime call (line 11) (which could still be scheduled to run on the current thread).

Fig. 7 also shows that even if the *immediate* path is much simpler than the *creation* path, it is not as simple as just calling the outline function. Proper implementation of OpenMP task semantics requires additional bookkeeping for **`firstprivate`** variables and for the context of inner tasks (lines 22 to 24). This bookkeeping becomes noticeable when the **`if`** clause is used to control or *aggregate* fine grain tasks, like in Fig. 1.

4.2 Adding Final Support

Adding support for the **`final`** clause to our existing implementation is straightforward. When the expression in the **`final`** clause evaluates to *true*, the task is flagged as *final*.

```
1 float g1(float);
2 float g2(float);
3 float g(float b)
4 {
5     float b1, b2;
6 #pragma omp task shared(b1)
7     b1 = g1(b);
8 #pragma omp task shared(b2)
9     b2 = g2(b);
10 #pragma omp taskwait
11     return b1 + b2;
12 }
13 float f(float a)
14 {
15     float b;
16 #pragma omp task if(if_expr) final(final_expr) shared(b) firstprivate(a)
17     if (a > 0)
18       b = a + g(a) + f(a - 1);
19
20 #pragma omp taskwait
21     return b;
22 }
```

Fig. 6. Example of task code

```
1 nth_creation_mode_res_t nth_creation_mode = CREATION_MODE_IMMEDIATE;
2 if ( if_expr )
3     nth_creation_mode = nth_creation_mode_create(); // Runtime call
4 switch (nth_creation_mode)
5 {
6   case CREATION_MODE_CREATE :
7     {
8       /* Create a task */
9       nth_desc * nth;
10      ... // Code that fills arguments of the outline
11      nth = nth_create_task_ci((void *) (nth__f_3), &nth_type, &nth_ndeps,
12          &nth_vp, &nth_succ, NTH_CI_ALL, &nth_arg_addr_ptr, &nth_nargs_ref,
13          &nth_nargs_val, &b, &nth_size[1], &a);
14      nth_submit(nth);
15      break;
16    }
17  case CREATION_MODE_IMMEDIATE:
18    {
19      /* Run the task immediately */
20      float cval_a = a;  /* firstprivate variable */
21      nth_task_ctx_t nth_ctx;
22      nth_push_task_ctx(&nth_ctx);
23      nth__f_3(&b, &cval_a);
24      nth_pop_task_ctx();
25      break;
26    }
27  default : { /* Invalid creation mode */ __builtin_abort(); break; }
28 }
```

Fig. 7. Implementation of the **`task`** construct in Nanos 4

Once a task is marked as *final*, the creation mode of any other task created by the *final task* will always be *immediate*.

This implementation, while fulfilling the semantics of the **`final`** clause, performs unnecessary work. As shown in Fig. 7 (lines 22 to 24) the *immediate* path still requires

```
1  if (NTH_MYSELF->task_ctx->is_final)
2  {
3    // Final creation mode path
4    nth__f_3(&b, &a);
5  }
6  else
7  {
8    nth_creation_mode_res_t nth_creation_mode = CREATION_MODE_IMMEDIATE;
9    if (expr)
10     nth_creation_mode = nth_creation_mode_create(); // Runtime call
11
12   char _task_is_final = (final_expr);
13   switch (nth_creation_mode)
14   {
15     case CREATION_MODE_CREATE :
16       {
17         /* Create a task */
18         nth_desc * nth;
19         ... // Code that fills arguments of the outline
20         nth = nth_create_task_ci( ... );
21         if (_task_is_final)
22           nth->task_ctx->is_final = 1;
23         nth_submit(nth);
24         break;
25       }
26     case CREATION_MODE_IMMEDIATE:
27       {
28         /* Run the task immediately */
29         float cval_a = a;
30         nth_task_ctx_t nth_ctx;
31         nth_push_task_ctx(&nth_ctx);
32         if (_task_is_final)
33           NTH_MYSELF->task_ctx->is_final = 1;
34         nth__f_3(&b, &cval_a);
35         nth_pop_task_ctx();
36         break;
37       }
38     default : { /* Invalid creation mode */ __builtin_abort(); break; }
39   }
40 }
```

Fig. 8. Implementation of the **`task`** construct supporting the **`final`** clause

some bookkeeping. This bookkeeping is only needed if a task that runs immediately creates new tasks. If one of these tasks does not run immediately, the task which created them needs to properly synchronize to fulfill OpenMP semantics. As an example of such case, consider the task in line 16 of Fig. 6. If the runtime chooses not to create that task, running it immediately, the task code can call function `g` which creates two tasks. That task in `f` needs to properly synchronize with those two tasks in case they do not run immediately. This bookkeeping is unnecessary in *final tasks*: all tasks created by a *final task* always run immediately.

So, we added an additional creation mode: *final*. When the creation mode is *final* the task runs as if the creation mode is *immediate* but the bookkeeping is avoided. Since a *final task* will never create tasks, regardless of the value of the **`if`** clause, we add an early check to determine if the creating task is itself *final*, as is shown in Fig. 8. On line 1 the creating task first checks if it is final. If it is *final*, then the task runs immediately (line 4). If it is not *final*, the task creation proceeds as it did previously (see Fig. 7).

We also protect the runtime calls that implement the **taskwait** construct with a check to the *final* flag. If the task is *final* the call will be omitted (as it will have no children tasks).

The implementation of the **omp_in_final** API call straightforwardly returns the value of the *final* flag.

4.3 Advanced Final Support

While the approach in section 4.2 harnesses the properties of *final tasks* to improve performance, we have not yet fully realized the benefits of *final tasks*. The handmade transformation from Fig. 2 controls both task granularity and overhead. A similar solution can be implemented automatically.

Since *final tasks* do not create any tasks, *final tasks* can be implemented by rewriting their code where all the tasking constructs (i.e., **task** and **taskwait**) have been removed. If we repeat this process recursively for all the functions called from the task code we get what we call the *serial closure*. This serial closure is, basically, a copy of all the code potentially called from a single task but without OpenMP tasking constructs and straightforwardly implements the *final task* semantics. We also substitute the calls to **omp_in_final** by the constant 1 as the task will always be final in the *serial closure*. This substitution enables further optimizations by the compiler backend (e.g., constant propagation, dead code elimination).

Fig. 9 shows how the **final** clause can be implemented using the serial closure. When the task is final (line 16), we invoke the serialized version of the task code. Since functions `f` and `g` are directly or indirectly called by the created task, a serial version of each without OpenMP (lines 1 and 8), is created and called instead of the original functions inside the serial closure (line 12).

```
1  float __serial_g_(float b)
2  {
3    float b1, b2;
4    b1 = g1(b);
5    b2 = g2(b);
6    return b1 + b2;
7  }
8  float __serial_f_(float a)
9  {
10   float b;
11   if (a > 0)
12     b = a + __serial_g_(a) + __serial_f_(a - 1);
13   return b;
14 }
15 ...
16 if (NTH_MYSELF->task_ctx->is_final)
17 {
18   // final closure path
19   // task code is inlined
20   if (a > 0)
21     b = a + __serial_g_(a) + __serial_f_(a - 1);
22 }
```

Fig. 9. Implementation of the **final** clause under serial closure

Implementing the **`final`** clause using the serial closure has the benefit that the overhead due to OpenMP runtime calls is minimized. This does not happen if we simply call the outline function of a *final task* (like in Fig. 8). Any task created inside that *final task* still has to check the *final* status.

Applicability of serial closure is limited by the availability of the code. Code in libraries usually cannot benefit from serial closure because most of the time the code is not available at compile time. In those cases, the tasks will have to resort to testing the creation mode. In Fig. 9, functions `g1` and `g2` are called directly (lines 4 and 5) because their code is not available. If these functions create tasks they will have to check the creation mode.

Code size constraints can also pose a problem to the serial closure approach, since depending on the program, a large amount of code may be duplicated, but that would also happen if a user did it manually as in Fig. 2.

5 Evaluation

5.1 Methodology

To evaluate the possible benefits of the **`final`** clause we have used three applications from the BOTS suite[5] that present fine grain parallelism: *nqueens*, *floorplan* and *fibonacci*. For each application we have executed different versions:

if-clause. In this version the granularity control of tasks is done by means of an OpenMP **`if`**, much as in Fig. 1.

manual-unoptimized. In this version the granularity control is manually introduced by the programmer modifying the source code, as in Fig. 2, with the serial path executing the same code as the parallel path.

manual-optimized. The same as the *manual-unoptimized* version but the unneeded code has been removed from the serial path. For *Fibonacci* the serial path cannot be optimized so we will refer to it just as *manual*.

final. In this version, the **`final`** clause controls the granularity of tasks using the first simple implementation described in Section 4.

final-closure. In this version, we also used the **`final`** clause but with the serial closure described in Section 4.

final-closure+api. In this version, we used the proposed **`omp_in_final`** API call in addition to the **`final`** clause to optimize the *final* path.

Comparing the *final* versions with the *if-clause* version will show how much benefit can be obtained by lifting some of the constraints of the **`if`** clause. By comparing the *final* versions against the *manual* versions we can compare the performance we obtain and the one programmer could obtain by modifying his code.

Finally, we compare the the *final* and *final-closure* implementations of the **`final`** clause to quantify which are the benefits (if any) of the more complex implementation of *final-closure* where there is also code replication (see Section 4).

All the benchmarks were compiled with an extended Mercurium C/C++ source-to-source compiler that implements the **`final`** clause for tasks targetting the Nanos 4 [8]

runtime. The backend compiler used by the Mercurium source-to-source compiler was `gcc` 4.1 with -O3 optimization level.

We executed the benchmarks on a SGI Altix 4700 with Itanium 2 processors using a *cpuset* configuration that avoids interferences with other threads in the system. For the purpose of this evaluation we used up to 32 processors.

For each experiment we have run 5 executions and we present in the next sections the average speed-up (and the mean deviation). For *nqueens* and *fibonacci* the speed-up we show is in execution time against the optimized serial version. In the case of *floorplan*, because the execution is not deterministic, the speed-up is computed using the number of tasks per second metric [5].

5.2 Results

NQueens. Fig. 10 shows the experimental results obtained with the four different versions to control granularity for the *NQueens* application with a 14x14 board size. As we saw in Section 2 the **`if`** version obtains a very limited speed-up compared to the *manual* versions. The results show that using the proposed **`final`** clause we see a significant performance increase with the simple version and even more with the version using the *serial closures*. While the *final-closure* version cannot achieve the performance of the *manual-optimized* version because it cannot optimize the serial path, it obtains a scalability on par with the *manual-unoptimized* version without modifying the original source at all.But, by using the proposed API call the user can optimize the path and obtain a performance on par with the *manual-optimized* version without having to split his code manually.

Fibonacci. Fig. 11 shows the performance results, for all different versions, to compute the 40th Fibonacci number. Fibonacci presents very fine grain tasks that make it very difficult to scale without aggregating them efficiently. The results show the version with **`if`** clause, because of its semantics, does not scale at all. The version using the

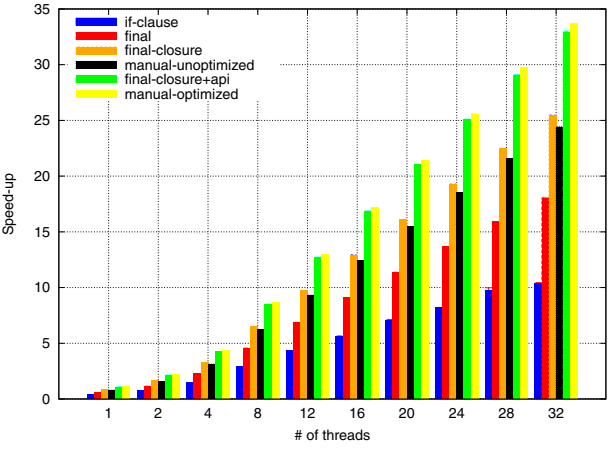


Fig. 10. Speed-up results for NQueens with a 14x14 board

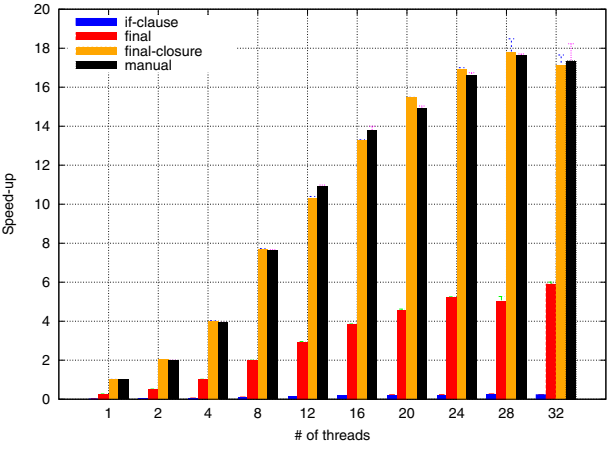


Fig. 11. Speed-up results for Fibonacci with input 40

simple **`final`** implementation already is able to scale a bit but because tasks are so fine grained even small overheads impacts the final performance. But, when we use the **`final`** version using the *serial closure* we obtain performance as good as the *manual* version, with a considerable gain over the simple **`final`** implementation.

Floorplan. Fig. 12 shows the results obtained for the *floorplan* benchmark with the input of 20 cells. Floorplan is a search algorithm with pruning. As such, it is very irregular. While the version with the **`if`** clause scales a bit, performance is far below the *manual* versions. Implementing it with the proposed **`final`** clause (with or without the serial closure) we obtain a version that performs as the *manual* without optimizatios. When we use the **`final`** API we obtain results on par with the *optimized manual* version.

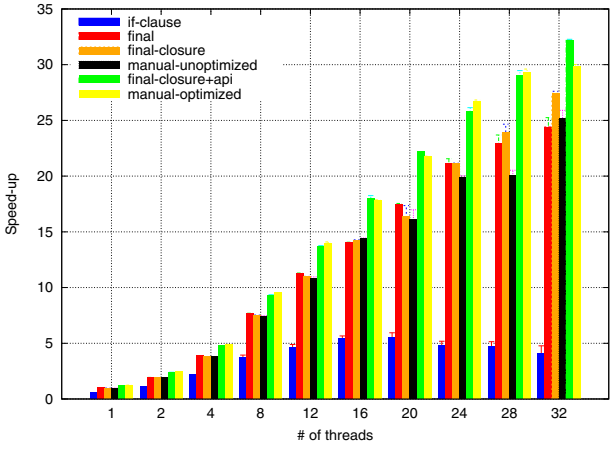


Fig. 12. Speed-up results for Floorplan with 20 cells

Code size analysis. As we mentioned in Section 4.2, the use of a serial closure can increase the size of the executable. Fig. 13 shows the size of the linked binary for each of the versions of the different applications. We can see that although the **`final`** versions with *closures* introduce a bit more of code than the *manual* version, these versions do not result in a code explosion. Better analysis of the code that needs to go into the closure could further reduce code duplication. So, overall the code increase does not pose a problem.

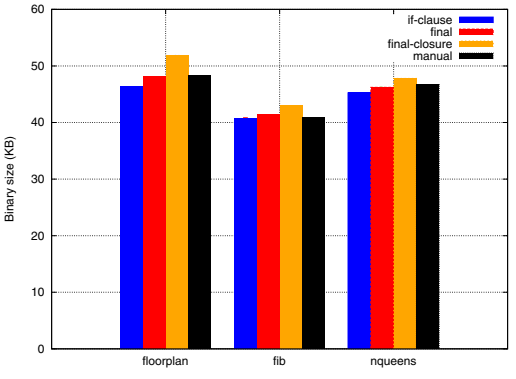


Fig. 13. Binary size of different versions

6 Conclusions and Future Work

Our evaluation shows that the final clause we propose can improve scaling of algorithms entailing a wide range of task granularity by reducing the overheads associated with execution of fine-grained tasks (particularly if serial closures and the **`omp_in_final`** API call are used). The clause and API call enable these gains under programmer control to a level that is very difficult to achieve by automated optimization.

The clause is a profound change to OpenMP in two respects. First, it affects whether data-sharing attribute clauses are honored or not. Second, it does so not for the construct it associates to, but for constructs encountered in its dynamic extent. Hence code semantics can change depending on the dynamic context, with significant impact on code development and programmer's approach. We are confident the first change is not harmful for *predetermined* or *implicitly determined* data-sharing attributes, but deeper investigation is needed for explicitly-determined ones. Both aspects need more investigations in terms of user experience.

Our proposal addresses recursively generated tasks for which the computational cost shrinks as depth increases. More investigation is needed to address cases where task generation is not recursive or cost is not related to generation depth.

Acknowledgements

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050), the Generalitat

de Catalunya (2009-SGR-980), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC Network of Excellence (contract FP7/ICT 217068), and the MareIncognito project under the BSC-IBM collaboration agreement. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

References

1. Corbalán, A.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)
2. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems 20, 404–418 (2008)
3. Brunschen, C., Brorsson, M.: Odin/CCp–A Portable Implementation of OpenMP for C. Concurrency: Practice and Experience. Special issue on OpenMP 12(12), 1193–1203 (2000)
4. Barcelona Supercomputing Center. The NANOS Group Site: The Mercurium Compiler, http://nanos.ac.upc.edu/mcxx
5. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: 38th International Conference on Parallel Processing (ICPP '09), Vienna, Austria, September 2009, pp. 124–131. IEEE Computer Society, Los Alamitos (2009)
6. Kapinos, P., an Mey, D.: Parallel simulation of bevel gear cutting processes with openMP tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 1–14. Springer, Heidelberg (2009)
7. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
8. Teruel, X., Martorell, X., Duran, A., Ferrer, R., Ayguadé, E.: Support for OpenMP tasks in Nanos v4. In: CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research, pp. 256–259. ACM, New York (2007)

Towards an Error Model for OpenMP

Michael Wong[1], Michael Klemm[2], Alejandro Duran[3], Tim Mattson[2], Grant Haab[2], Bronis R. de Supinski[4], and Andrey Churbanov[2]

[1] IBM Corporation
[2] Intel Corporation
[3] Barcelona Supercomputing Center
[4] Lawrence Livermore National Laboratory
michaelw@ca.ibm.com, alex.duran@bsc.es, bronis@llnl.gov,
{michael.klemm,timothy.g.mattson}@intel.com,
{grant.haab,andrey.churbanov}@intel.com

Abstract. OpenMP lacks essential features for developing mission-critical software. In particular, it has no support for detecting and handling errors or even a concept of them. In this paper, the OpenMP Error Model Subcommittee reports on solutions under consideration for this major omission. We identify issues with the current OpenMP specification and propose a path to extend OpenMP with error-handling capabilities. We add a construct that cleanly shuts down parallel regions as a first step. We then discuss two orthogonal proposals that extend OpenMP with features to handle system-level and user-defined errors.

1 Introduction

OpenMP [14] is a wide-spread and well-known programming model for parallel programming on shared memory platforms. OpenMP's initial focus was to provide portable parallel programming for High Performance Computing (HPC) platforms. OpenMP's expressive programming model, including support for incremental parallelization, has led application programmers in other areas (e. g., enterprise software) to consider it for their applications. However, OpenMP's lack of any concept of errors or support to handle them has prevented wide-spread adoption of OpenMP by industries.

OpenMP 3.0 only requires an implementation to provide best effort execution for runtime errors. Application are often terminated and users must restart. While perhaps tolerable even if undesirable for HPC users, it is clearly unacceptable to terminate enterprise applications. Thus, programmers must implement workarounds (such as those in Section 2) that make development and maintenance more difficult and often prevent key compiler optimizations.

In this paper, we present the current plans of the OpenMP Error Model Subcommittee to provide error handling extensions. Clean semantics for errors raised in concurrent code paths are non-trivial [8,18,20] so we do not focus on solutions to concurrent errors. We instead consider mechanisms to detect and to respond to errors (both OpenMP runtime and user code errors). We provide our criteria and limitations for OpenMP error proposals in Section 3. We then propose a two-phase process to add error support in

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 70–82, 2010.

```
1 void f(float *x, int am, int points);
2
3 void a(float *x, int np) {
4     int am, points;
5     omp_set_dynamic(0);
6     omp_set_num_threads(16);
7 #pragma omp parallel shared(x, np) private(am, points)
8     {
9         if (omp_get_num_threads() != 16)
10            abort();
11        am = omp_get_thread_num();
12        ...
13        f(x, am, points);
14    }
15 }
```

Fig. 1. Setting and checking the threads count for parallel regions

Section 4. We plan to add a `done` construct that cleanly terminates an OpenMP region to the next OpenMP version while our longer term strategy is considering two proposals of catching and handling OpenMP runtime and user-defined errors.

2 Current State in OpenMP Error Handling

This section motivates the need for clean error handling semantics in OpenMP. We first investigate the current OpenMP specification's error handling requirements for errors that arise within OpenMP implementations. We then turn to OpenMP's features for handling user-defined errors, i. e., C++ exceptions and return codes in C and Fortran.

2.1 OpenMP Runtime Errors

OpenMP has never offered clean semantics to handle errors that arise within OpenMP runtime implementations. Fig. 1 shows a similar example to Example 40.1.c, which we chose arbitrarily from the OpenMP 3.0 specification [14]. We use this example to investigate potential errors that may arise and how OpenMP deals with them.

The code in Fig. 1 calls `omp_set_dynamic` (line 5) and `omp_set_num_threads` (line 6) to ensure that exactly 16 threads execute the following parallel region. OpenMP does not prescribe how an OpenMP implementation should react if a programmer passes inconsistent values (e. g., a negative thread count) to these functions. Any behavior, from terminating the application to using any (valid) value is compliant.

If the OpenMP runtime cannot supply the requested number of threads (e. g., due to resource constraints) for the parallel region, OpenMP does not prescribe how the implementation must react. For instance, it can terminate the program with (or without) an error message or continue with an arbitrary thread count. Thus, programmers must explicitly check the thread count by calling `omp_get_num_threads` and take appropriate actions; in this example, the program explicitly aborts (line 10) if it does not get exactly 16 threads.

Other errors can occur when the OpenMP runtime creates the threading environment and allocates resources for the parallel region. The OpenMP specification allows the

```
1 void example() {
2     try {
3 #pragma omp parallel
4         {
5 #pragma omp for
6             for (int i = 0; i < N; i++) {
7                 potentially_causes_an_exception();
8             }
9             phase_1();
10 #pragma omp barrier
11             phase_2();
12         }
13     }
14     catch (std::exception *ex) {
15         // handle exception pointed to by ex
16     }
17 }
```

Fig. 2. Non-conforming OpenMP code

OpenMP implementation to define how it responds to these errors. Hence, programmers cannot intercept potential errors in a portable way in order to take more appropriate actions.

OpenMP runtime support routines also have no defined error semantics. The specification does not prescribe actions for an OpenMP implementation if routines fail, which is also true if users supply incorrect values for OpenMP environment variables. These unspecified behaviors complicate the implementation of resilient applications, which must continue functioning in the presence of errors or unexpected conditions.

2.2 User-Defined Errors

Most mission-critical applications cannot silently ignore errors and continue execution. Thus, error handling consumes significant application development time [16]. We now discuss how OpenMP applications must handle C++ exceptions; we cover C and Fortran error handling patterns at the end of this sub-section.

Sequential C++ codes usually map errors to exceptions that are thrown where the error arises and caught by error handling code. Fig. 2 shows a simple program skeleton that does not conform to the OpenMP specification if an exception arises in the `for` construct. Several threads that execute could raise an exception and concurrent exceptions could occur. Any exception would cause a premature termination of the parallel region, which violates the Single-Entry Single-Exit (SESE) principle that is required of all OpenMP parallel regions including those that use `longjmp()` or `throw` [14]. Exceptions also must not escape any worksharing region or critical or master section. Thus, applications must catch all exceptions thrown within any structured block that is associated with an OpenMP construct before the block is exited. A throw that is executed inside an OpenMP region must cause execution to resume within the same region and the same thread that threw the exception must catch it.

Fig. 3 shows how to handle exceptions in a parallel region correctly by catching exceptions (line 19) so that they do not escape an OpenMP construct. For simplicity, we assume that all potential exceptions inherit from the C++ standard exception class

```
1 void example() {
2     std::exception *ex = NULL;
3 #pragma omp parallel shared(ex)
4     {
5         ...
6 #pragma omp for
7         for (int i = 0; i < N; i++) {
8             // if an exception occurred, cease execution of the loop body
9             // (the 'if' effectively prohibits most compiler optimizations)
10 #pragma omp flush
11             if (!ex) {
12                 // catch a potential exception locally
13                 try {
14                     potentially_causes_an_exception();
15                 }
16                 catch (const std::exception *e) {
17                     // remember to handle it after the parallel region
18 #pragma omp critical
19                     ex = e;
20                 }
21             }
22         }
23 #pragma omp flush
24         // if an exception occurred, stop executing the parallel region
25         if (ex) goto termination;
26         phase_1();
27 #pragma omp barrier
28         phase_2();
29     termination:
30         ;
31     }
32     if (ex) {
33         // handle exception pointed to by ex
34     }
35 }
```

Fig. 3. Shutting down an OpenMP-parallel region in presence of an exception

`std::exception`. We use a shared variable to notify other threads that the exception occurred. As we cannot prematurely terminate the `for` construct (the OpenMP specification prohibits changing the loop control variable or using a `break` statement), we use the exception flag to skip the remaining loop body when an exception occurs; this conditional `if` disables many standard compiler optimizations such as vectorization. Thus, the compliant code will run slowly even if no exceptions arise. We check the flag again before `phase_1()` is executed to ensure that we skip line 26 branching to the `termination` label (line 29). We again check the flag (line 32) following the parallel region to handle the exception consistently with the sequential semantics.

While Fig. 3 provides a method to terminate a parallel region, worksharing constructs and tasks require conditional tests to skip the remainder of the structured block. Synchronization constructs such as critical can use the C++ RAII (Resource Acquisition is Initialization) idiom, which scopes locks to ensure they are properly released. However, we know of no suitable workarounds for the `sections` and `ordered` constructs. As OpenMP `section` constructs are statically defined tasks that are executed concurrently, intercepting execution of `section` constructs in a conforming way (e. g., by setting a `shared` variable) is impossible. An `ordered` construct serializes part of a loop's execution. While it is possible to terminate the current `ordered` construct,

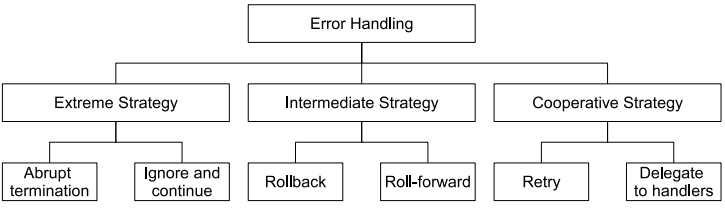


Fig. 4. Classification of error handling strategies

programmers cannot influence execution of other `ordered` constructs. Avoiding execution of subsequent `ordered` constructs effectively leads to the loop-termination pattern of Fig. 3.

Most applications in C and Fortran indicate errors with special return values or an `errno` variable (e. g., POSIX return codes [19]) or through an additional function argument (e. g., MPI error codes [12]). Programmers must check these manually to determine if an error has arisen and handle it similarly to the coding pattern shown in Fig. 3. Nonetheless, OpenMP SESE requirements often cause substantial changes to the sequential coding pattern in these languages.

3 Design Objectives

We now discuss the major requirements for the cleaner error handling semantics that OpenMP clearly needs. First, the new error model must support all OpenMP base languages and allow for additional ones consistently with the OpenMP philosophy of incremental parallelization. Thus, it must support the methods of exception-aware (e. g., C++) and exception-unaware (e. g., C and Fortran) languages. The error handling facilities must tightly integrate modern exception handling while supporting traditional techniques based on status flags and return values in order to avoid significant changes to the sequential code.

Second, the new error model must provide exception-aware support even for exception-unaware languages since they have cleaner, stronger semantics than classic mechanisms (e. g., error codes). Programs cannot ignore (or forget to detect) exceptions, which always force the programmer to respond in some way. Return codes intermix data and errors, while exceptions decouple error handling from the primary application functionality. Finally, exceptions simplify the reuse of error handling code and eliminate repetitive, error-prone code since programmers do not have to locate error handlers where errors arise.

The new model must support system-level errors as well as user-defined ones. Ideally, an OpenMP implementation must provide notification of errors that arise within the runtime system without requiring special checks. Further, the model must not require the programmer to distinguish between system-level and user-defined errors although it should allow the user to react differently depending on the source of the error. However, programmers should rely on a single, common framework that can handle errors from either source.

The model must be flexible and provide tools to implement different error handling strategies. Fig. 4 classifies error-handling mechanisms into different categories. Our goal is to support the *extreme strategy* and *cooperative strategy*. Intermediate strategies are beyond the scope of our working group. However, they could be implemented through transactional memory [10], which is a possible OpenMP extension [11].

Most importantly, the error handling features must provide backward compatibility. It cannot break existing codes that have adapted to the current "best effort" error-handling requirements. It also must not require new applications to handle errors if they do not involve mission-critical computations.

4 OpenMP Error Handling Proposals

We propose a two-phase plan to satisfy our error model requirements. Our first step will provide a construct to terminate OpenMP regions, which directly supports the *abrupt termination* pattern for user-defined errors of Fig. 4. Section 4.1 describes the `done` construct that the OpenMP Error Model Subcommittee has proposed for the OpenMP 3.1 specification.

Our second step will support the *ignore and continue*, *retry* and *delegate to handlers* strategies. We describe the two orthogonal proposals that the OpenMP Error Model Subcommittee is considering in Section 4.2 and Section 4.3. We discuss the implications of these mechanisms, for which we target OpenMP 4.0, with a specific focus on backwards compatibility.

4.1 The `done` Construct

Our proposed `done` construct terminates innermost OpenMP regions, which provides initial support for user-defined errors (particularly for C and Fortran). We build on prior proposals to terminate parallel regions that were considered for the OpenMP 1.0 specification. HP's compiler still supports a limited `pdone` construct [1]. The `done` construct reflects the OpenMP philosophy of incremental parallelization through compiler directives, unlike the alternative of a new runtime function, which would alter the underlying sequential code or require conditional compilation guards.

The `done` construct has the syntax[1]

```
1 #pragma omp done [clause-list]
```

with *clause-list* being one or more of `if(expr)`,[2] `parallel`, `for` (C/C++), `do` (Fortran), `sections`, `single`, or `task`. The binding set of the `done` construct is the current thread team. It applies to the innermost enclosing OpenMP construct(s) of the types specified in the clause (i.e., `parallel` or `task`).

The `done` construct causes the encountering thread to terminate the subset of the innermost parallel, worksharing and task regions that corresponds to the specified clause. It conceptually sets a *cancellation flag* that the other team members must evaluate at no later than the next *cancellation point* that they execute. Hence, other threads may not

[1] We present the C/C++ syntax only; the Fortran syntax is similar.

[2] The `if` clause has the same semantics as the `if` clauses of other constructs.

```
1  void example() {
2      std::exception *ex = NULL;
3  #pragma omp parallel shared(ex)
4      {
5          ...
6  #pragma omp for
7          for (int i = 0; i < N; i++) {
8              // no 'if' that prevents compiler optimizations
9              try {
10                 causes_an_exception();
11             }
12             catch (const std::exception *e) {
13                 // still must remember exception for later handling
14 #pragma omp critical
15                 ex = e;
16 #pragma omp done parallel for
17             }
18         }
19         phase_1();
20 #pragma omp barrier
21         phase_2();
22     }
23     // continue here if an exception is thrown in the 'for' loop
24     if (ex) {
25         // handle exception stored in ex
26     }
27 }
```

Fig. 5. Shutting down an OpenMP parallel region with the `done` construct

immediately terminate execution when a thread encounters the `done` construct. This delayed termination allows more efficient execution, as the mechanism does not require interrupts or frequent polling of the cancellation flag.

We make the set of cancellation points implementation defined in order to avoid restricting implementation choices although we are exploring a minimal set. Efficient implementation of the `done` construct will likely require different cancellation points under different OpenMP implementations. A minimal set of cancellation points could be: entry and exit of regions, barriers, critical sections, completion of a loop chunk and calls to runtime support routines.

The `done` construct supports elegant and robust termination of OpenMP parallel execution, as Fig. 5 shows. If an exception is raised during execution of the worksharing construct from lines 6–18, the catch handler can trigger a `done parallel for` to shut down the worksharing construct safely. The `for` clause terminates the worksharing construct while the `parallel` clause terminates the parallel region. Execution continues at the `if` in line 24 after termination of the regions. While the `done` construct cleanly terminates OpenMP regions, programmers must still track exceptions through pointers and apply the sequential handler. However, there is no need for the tricky `flush` as the exception pointer is not accessed within the parallel region.

4.2 Proposal Based on return Codes

In the long term, we must provide features that support the existing error-handling code in exception-unaware languages. We also must ensure backwards compatibility for

existing OpenMP programs, which limits the mechanisms and API choices for our error-handling model.

Thus, we consider the minimal functionality that an error-handling system requires. An OpenMP error-handling mechanism must:

- Communicate to the user program that an error has occurred;
- Provide sufficient information to identify the type and source of the error.
- Support execution after the error arises with well-defined program state so that the program can respond and continue.

We now discuss a modest error-handling proposal based on return codes that meets these requirements.

In order to support continued execution, we require that the program continues at the first statement following the end of the innermost construct when an error occurs inside any OpenMP construct. Any variables that are created or modified inside the construct have an undefined value.

We communicate the error condition to the program through a variable that is shared between the members of the thread team. The `omp-error-var` variable is of type `omp_error_t` and stores an error code that identifies whether any thread that executed the preceding OpenMP construct or runtime library routine encountered an error and, if so, the error's type. If concurrent errors occur, the runtime system may arbitrarily select one error code and store it in the shared variable.

Programs can query the value of this variable by calling a new OpenMP runtime support routine:

```
1 omp_error_t omp_get_error(char * omp_err_string, int bufsize)
```

This function can return any value of a set of constants that are defined in the standard OpenMP include file. While implementations can support additional error codes, we anticipate a standard set that may include values such as:

- `OMP_ERR_NONE`
- `OMP_ERR_THREAD_CREATION`
- `OMP_ERR_THREAD_FAILURE`
- `OMP_ERR_STACK_OVERFLOW`
- `OMP_ERR_RUNTIME_LIB`

The `omp_get_error` function in addition returns an implementation-defined, zero-terminated string in the memory area pointed to by `omp_err_string`. In the string, an implementation may provide more information about the type and source of the error.

Fig. 6 shows an example that uses our error code handling proposal. While the code is less elegant than approaches based on exceptions and callbacks, we can add this solution to any OpenMP base language and satisfy our backward compatibility requirement. This error code approach is more straightforward in that it does not introduce complex execution flows through error handlers. This solution is directly based on typical error handling in C and Fortran.

```
1 #include "omp.h"
2
3 #define BUFSIZE ...
4
5 void error_res(omp_error_t perr, char *err_str) {
6   // User-written function to clean up, report,
7   // and otherwise respond to the error
8 }
9
10 int main {
11     omp_error_t perr;
12     char * err_str;
13     int terminate = 0, nths = 16;
14     while (!terminate) {
15 #pragma omp parallel numthreads(nths)
16         {
17             ... // The body of the region
18         }
19         if ((perr = omp_get_error(err_str, BUFSIZE)) != OMP_ERR_NONE) {
20             error_res(perr, err_str);
21             if(perr == OMP_ERR_THREAD_CREATE) {
22                 nths  = (nths > 1) ? (nths -1) : 1;
23             }
24             else {
25                 printf("unrecoverable error\n");
26 #pragma omp critical
27                 terminate = 1;
28             }
29         }
30     }
31 }
```

Fig. 6. Using the proposal based on error codes

4.3 The Callback Error Handling Mechanism

Our longer term proposal supports callbacks in the event of an error. It provides many of the benefits of exception-aware languages to C and Fortran. This proposal, which slightly extends a previous callback-based proposal by Duran et al. [3], achieves its functionality by means of *callback* notifications and supports both exception-aware and exception-unaware languages. Hence, it supports all OpenMP base languages.

The prior proposal extends OpenMP constructs with an `onerror` clause that overrides OpenMP's default error-handling behavior, as line 8 of Fig. 7 shows. Programmers specify a function that is invoked if any errors arise (lines 1–4) within the OpenMP implementation (e. g., directives and API calls). The handler can take any necessary actions and notify the OpenMP runtime about how to proceed with execution (e. g., retry, abort, or continue). The prior proposal also provided a set of default handlers that the program can specify with the `onerror` clause to implement common error responses. Also, the `context` directive associates error classes and error handlers with sequential code regions to support errors that arise in OpenMP runtime routines. Users are not required to define any callbacks in which case the implementation will provide backward compatibility with the current *best effort* approach.

Our callback proposal extends the `onerror` proposal to meet our OpenMP error-handling model requirements. We add the error class `OMP_USER_CANCEL` to associate error handlers with termination requests of `done` constructs, which supports voluntary

```
1 omp_error_action_t savedata(omp_err_info_t *error, my_data_t *data) {
2     /* save computed data */
3     return OMP_ABORT; // notify the resolution to the error
4 }
5
6 void f() {
7     my_data_t data;
8 #pragma parallel onerror(OMP_ERR_SEVERE,OMP_ERR_FATAL: savedata, &data)
9     {
10        /* parallel code */
11    }
12 }
```

Fig. 7. Example of OpenMP error handling using the callback proposal

region termination and, thus, user-defined error handling. We provide the error class `OMP_EXCEPTION_RAISED`, so that error handlers can catch and handle C++ exceptions, either locally or globally by re-throwing. Thus, this mechanism supports the exception-aware semantics of C++ that handle and re-throw exceptions. Finally, we are exploring extensions such as specifying a default handler with an environment variable so that applications can take appropriate actions for errors that occur during initialization of the OpenMP runtime or from invalid states of internal control variables.

5 Other Concurrent Programming Error Handling Models

POSIX threads (pthreads) [6] is one of the earliest concurrent programming models. It specifies a binding for C, which all vendors have reused for C++ and Fortran. An effort is currently underway to define a POSIX threads binding for C++ [17]. Four error handling models [5] are commonly used for threads; two apply to pthreads.

The first, and the simplest pthreads model, stops a thread when an error occurs using `pthread_kill` or `pthread_cancel` for asynchronous mode. This method unconditionally stops a thread and provides no mechanism to respond by calling cleanup actions. The Java^{TM} Thread API [13] includes a similar facility, `Thread.destroy` or `Thread.stop`. This model was reasonable for exception-unaware languages but does not support exception-aware languages. It could corrupt a running program because a thread may be partially completed, leaving some object or data in an incomplete state. Although incomplete, any programming model should support this simplest termination error handling model, which is why we propose the `done` construct.

The second model allows the target thread to delay its response to a termination request. The many examples of this model all use the idea of an interruption or cancellation point that is a well-defined point at which the target thread must respond. The full list of pthreads interruption points include calls to `wait()`, `sleep()`, `create()` and `write()`. In addition, cancellation points are usually encountered when a thread is blocked while sleeping, joining another thread or task, or waiting for a mutex, semaphore signal or synchronization. When a thread requests another thread to cancel itself (with `pthreads_cancel` in pthreads), the request is mapped in the target thread to an exception that is checked at cancellation points and can be rethrown or handled. With

pthreads, a program can install a chain of cancellation handlers, which serve a similar purpose as destructors in C++.

With pthreads, cancellation requests cannot be ignored or caught: the target must stop at its next cancellation point and cancellation cannot be stopped once it has begun. While acceptable for C or Fortran, which lack exceptions and object destructors, exception-aware languages like C++ require the ability to catch and recover from errors and to continue correct execution. Since an OpenMP error handling model must support C++, OpenMP requires a richer model than pthreads.

The third model, a (partially) cooperative model, implements error interrupts as exceptions thrown by `wait`/`join`/`sleep` calls. Like pthreads, the target thread can let destructors unwind the stack and exit. Unlike pthreads, the target thread can choose to unwind its stack until it finds a handler that catches and handles the exception, and then resume normal operation. Alternatively, it can catch the exception immediately and ignore it. `Thread.interrupt` in Java and `Thread.Interrupt` in .NET provide a partial cooperative model, while one is currently being prepared in C++0x [4].

The fourth model, a fully cooperative model, allows the target thread to check whether it is the target of an interrupt anywhere, not just at cancellation points. This cooperative model is planned for C++0x.

MPI supports error handling with a callback mechanism [12]. This mechanism provides a default behavior of aborting when errors occur within MPI as well as an additional predefined handler that allows errors to return an error code. MPI error support is similar to our overall proposal although the issues are simpler since MPI defines error handling as a local operation other than with the default abort behavior.

Michael Süß [15] proposes a cancellation proposal with a pragma-based user technique for stopping threads. This proposal does not define any cancellation points, and is generally considered a cooperative exception approach. It is also limited to threads and regions and cannot shutdown other constructs, e. g., tasks or subteams. Our `done` construct is similar but simpler to use and covers all existing OpenMP constructs.

OpenMP for Java [9], JCilk [2], and TBB [7] support exception handling by setting an internal flag. OpenMP for Java and JCilk check this flag at cancellation points, terminate parallel execution, and rethrow exceptions to the sequential code. Both languages arbitrarily select one exception if multiple ones arise. TBB registers the first exception and cancels the task group; it ignores other concurrent exceptions. TBB supports exception propagation for C++0x to pass exceptions to other threads. All three models lack a flexible mechanism to react to error situations other than terminating parallel execution. With our proposal we strive to provide a toolbox of error-handling mechanisms that help programmers implement more sophisticated error-handling strategies.

6 Conclusions and Future Work

We have presented the current directions that the OpenMP Error Model Subcommittee is pursuing. OpenMP currently has no concept of errors. We have identified the requirements of OpenMP error-handling models. The most important requirements are the need to support all OpenMP base languages and to provide backward compatibility for applications that assume no error-handling support is available. The first requirement

means that the mechanism must not require significant changes to sequential code for exception-unaware and exception-aware languages.

We have detailed planned error-handling extensions for future OpenMP specifications. Our plans include the standardization of a `done` construct that supports termination of OpenMP regions, which not only supports error handling but will also prove useful for some task-based programs. We anticipate that OpenMP 3.1 will include this construct while we target OpenMP 4.0 for more complete error handling capabilities. For these more complete capabilities, we are investigating a proposal based on error codes that follows typical error handling in C and Fortran and a callback proposal that provides at least a partial cooperative model.

We still have issues to resolve for the 4.0 proposals. We are exploring a minimal set of required cancellation points. We also must integrate the proposals into the overall standard. For example, barriers can cause deadlocks in the presence of exceptions when all threads besides the one that catches an exception wait at a barrier. This behavior is technically non-conforming code; we must resolve this inconsistency, possibly through the minimal cancellation point set. Overall, we will continue to explore these issues and to design solutions that will provide a complete OpenMP error-handling model.

Acknowledgments

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-CONF-426251).

References

1. Compaq Computer Corporation. Compaq Fortran – Language Reference Manual (September 1999); Order number AA-Q66SD-TK
2. Danaher, J.S., Angelina Lee, I.-T., Leiserson, C.E.: Programming with Exceptions in JCilk. Science of Computer Programming 63(2), 147–171 (2006)
3. Duran, A., Ferrer, R., Costa, J.J., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: A Proposal for Error Handling in OpenMP. Intl. Journal of Parallel Programming 35(4), 393–416 (2007)
4. Becker, P. (ed.): Working Draft: Standard for Programming Language C++ (November 2009); Document number N3000

5. Sutter, H.: Interrupt Politely. Technical report (April 2008)
6. IEEE. Threads Extension for Portable Operating Systems (Draft 6) (February 1992); Document P1003.4a/D6
7. Intel Corporation. Intel Threading Building Blocks Reference Manual. Technical report, Document number 315415-003US (July 2009)
8. Issarny, V.: An Exception Handling Model for Parallel Programming and its Verification. In: Proc. of the Conf. on Software for Citical Systems, New Orleans, LA, USA, December 1991, pp. 92–100 (1991)
9. Klemm, M., Veldema, R., Bezold, M., Philippsen, M.: A Proposal for OpenMP for Java. In: Proc. of the Intl. Workshop on OpenMP, Reims, France (June 2006)
10. Larus, J.R., Rajwar, R.: Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, San Francisco (January 2007)
11. Milovanović, M., Ferrer, R., Unsal, O., Cristal, A., Martorell, X., Ayguadé, E., Labarta, J., Valero, M.: Transactional Memory and OpenMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 37–53. Springer, Heidelberg (2008)
12. Forum, M.P.I.: MPI: Extensions to the Message-passing Interface, Version 2.2. Technical report, MPI Forum (September 2009)
13. Oaks, S., Wong, H.: Java Threads, 3rd edn. O'Reilly, Sebastopol (2004)
14. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
15. Süß, M., Leopold, C.: Implementing irregular parallel algorithms with openMP. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 635–644. Springer, Heidelberg (2006)
16. Sommerville, I.: Software Engineering, 8th edn. Pearson Education, Ltd., Harlow (April 2007)
17. Stoughton, N.: POSIX Liaison Report, Document number N2536 (February 2008)
18. Tazuneki, S., Yoshida, T.: Concurrent Exception Handling in a Distributed Object-Oriented Computing Environment. In: Proc. of the 7th Intl. Conf. on Parallel and Distributed Systems Workshops, Iwate, Japan, July 2000, pp. 75–82 (2000)
19. The Open Group. The Open Group Base Specifications Issue 7 (December 2008); IEEE Std 1003.1-2008 and POSIX.1
20. Xu, J., Romanovsky, A., Randell, B.: Concurrent Exception Handling and Resolution in Distributed Object Systems. IEEE Transactions on Parallel and Distributed Systems 11(10), 1019–1032 (2000)

How OpenMP Applications Get More Benefit from Many-Core Era

Jianian Yan, Jiangzhou He, Wentao Han, Wenguang Chen, and Weimin Zheng

Department of Computer Science and Technology,
Tsinghua University, China
{yanjn03,hejz07,hwt04}@mails.tsinghua.edu.cn,
{cwg,zwm-dcs}@tsinghua.edu.cn

Abstract. With the approaching of the many-core era, it becomes more and more difficult for a single OpenMP application to efficiently utilize all the available processor cores. On the other hand, the available cores become more than necessary for some applications. We believe executing multiple OpenMP applications concurrently will be a common usage model in the future. In this model, how threads are scheduled on the cores are important as cores are asymmetric. We have designed and implemented a prototype scheduler, SWOMPS, to help schedule the threads of all the concurrent applications system-widely. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application provided by users. Experiment evaluation shows SWOMPS is quite efficient in improving the performance.

With the help of SWOMPS, we compared exclusive running one application and concurrent running multiple applications in term of system throughput and individual application performance. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual applications in concurrent execution is reasonable.

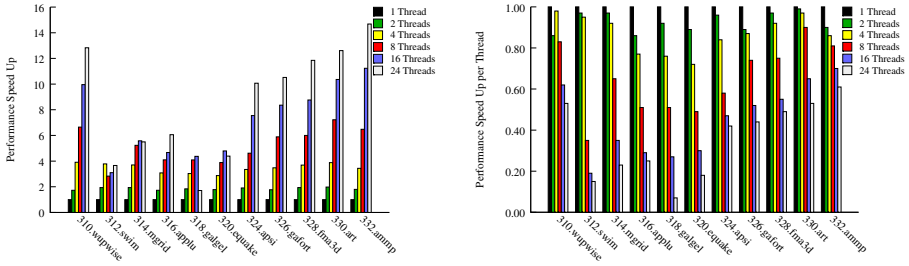
1 Introduction

Restricted by heating and power consumption, hardware vendors stopped increasing processor's performance by introducing complex circuit and increasing frequency. Instead, multiple cores are put into one chip. The effect of Moore's law has converted from increasing the performance of a single-core processor to the number of cores in a processor. Six-core general-purpose processor has been available in the market. Processors with more and more cores are coming soon. In the near future, one can easily have a computer with hundreds of cores by configuring multi-core processors in NUMA architecture.

In the multi-core era, OpenMP applications face the challenge of how to efficiently utilize the increasing computing power. Keeping the performance of OpenMP applications scaling with the number of processor cores is not trivial. Simply increasing the number of threads in an OpenMP applications does not guarantee the performance scaling. Figure 1 shows the result of scalability experiment with benchmarks in SpecOMP 2001. The experiment platform has 24 cores which detail configuration can be found in Sect. 4. In the test, the performance speedup per thread continues decreasing as the

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 83–95, 2010.

number of threads grows, as showed in Fig. 1b. And for some benchmarks, for example *314.mgrid*, *318.galgel* and *320.equake* showed in Fig. 1a, performance with 24 threads is even worse than performance with 16 threads. Writing well scaling applications requires sophisticated techniques and wide range of knowledge from hardware architecture to application domain. And theoretically, according to Amdahl's law, even for perfectly written applications, the existence of serialized portion in the application limits the benefit that can be obtained by parallel execution, no matter how many cores are available.



(a) Performance Speed Up (b) Performance Speed Up per Thread

Fig. 1. Performance Scaling with Various OMP Threads

Single OpenMP application has difficulty to efficiently utilize the increasing processor cores. On the other side, for some applications, it is unnecessary to utilize as many as the available cores. We believe multiple OpenMP applications executing concurrently will be a common usage model in the future. In this model, which cores the threads of an application are executing on will be an important factor to the performance. In a computer with multi-core processors configured in NUMA architecture, the cores are asymmetric. Cores in the same processor share the last level cache and have the same local memory while cores in different processors do not. An improper scheduling of threads on the cores will harm the performance of not only the application itself, but also other applications concurrently running on the computer.

We have designed and implemented a prototype scheduler SWOMPS (**S**ystem-**W**ide **O**pen**MP** application **S**cheduler) to help schedule the threads of all concurrent applications system-widely. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application provided by users. It will dynamically schedule the threads whenever an application comes to run or exits.

With the help of SWOMPS, we compared exclusive running one application and concurrent running multiple applications in term of system throughput and individual application performance. We take two application concurrently running for example. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual application in concurrent execution is reasonable.

The rest of the paper is organized as follows: Section 2 introduces two observations from the practical experience with OpenMP applications that guide the scheduler

design. Section 3 introduces the design and implementation of SWOMPS scheduler. Section 4 evaluates the scheduler and Sect. 5 compares the exclusive running model and the concurrent running model. Related work is introduced in Sect. 6 and Sect. 7 concludes the paper.

2 Practical Observations

2.1 Binding a Thread to a Core Will Improve Application Performance

Modern operating systems provide system calls with which user can specify a core that a thread is running on, which is also known as binding a thread to a core. OpenMP programmer usually binds each thread to a distinct core which will results in a better performance. Table 1 gives the execution time of the benchmarks with and without thread binding. Each benchmark is compiled with Open64 at O3 optimizing level, and running with 24 threads. The detail of the experiment environment can be found in Sect. 4. The last column of Table 1 shows the performance improvement due to thread binding. As we can see, the performance improvement ranging from 0.2% to 31.4%. There are two major sources of the performance improvement. Binding a thread avoids cache warming up when it is scheduled to a new core. And binding a thread keeps a core close to the data that it operates in the NUMA architecture.

Table 1. Execution Time Comparison of Thread Binding

Benchmark	Execution Time (s)		Perf. Impr.
	No Binding	Binding	
310.wupwise	190.13	170.36	11.6%
312.swim	405.21	401.94	0.8%
314.mgrid	660.11	523.93	26.0%
316.applu	228.76	227.93	0.4%
318.galgel	1192.54	939.25	27.0%
320.equake	148.59	121.95	21.8%
324.apsi	118.91	112.86	5.4%
326.gafort	311.10	310.53	0.2%
328.fma3d	222.95	199.97	11.5%
330.art	179.70	136.79	31.4%
332.ammp	320.40	302.94	5.8%

Table 2. Scheduling Preference of Each Benchmark

Benchmark	Execution Time (s)		Perf. Impr.
	Scatter	Gather	
310.wupwise	259.15	324.8	25.33%
312.swim	363.75	780.18	114.48%
314.mgrid	482.39	908.16	88.26%
316.applu	254.71	395.16	55.14%
318.galgel	616.53	676.9	9.79%
320.equake	129.54	179.15	38.30%
324.apsi	194.48	179.64	8.26%
326.gafort	479.53	641.55	33.79%
328.fma3d	290.59	377.77	30.00%
330.art	206.66	244.03	18.08%
332.ammp	510.3	490.78	3.98%

2.2 Different Applications Have Different Scheduling Preferences

Nowadays, most shared memory computers are configured in NUMA architecture. A processor can access its own *local memory* faster than non-local memory. A processor again contains several cores and cores in the same processor usually share the last level cache. The processor cores are asymmetric. Different OpenMP applications may have different thread scheduling preferences on the cores. Table 2 gives the results of a

scheduling preference experiment. The experiment runs on a computer with four processors, and each processor has six cores. A processor has 6M L3 cache that is shared amoung the 6 cores. In the experiment, each benchmark runs with 12 threads. In the *Scatter* scheduling, the 12 threads are scheduled onto 4 processors, each processor has 3 threads running on it. While in the *Gather* scheduling, the 12 threads are scheduled onto 2 processors, each processor has 6 threads running on it. In the scatter scheduling, the amount of shared cache per thread is larger and so are the memory bandwidth to local memory. In the gather scheduling, communication has lower cost as more threads are sharing the L3 cache, and more cores are sharing the same local memory that reduces the chance to access remote memory. Data in the *Perf. Impr.* column in Tab. 2 gives the performance improvements if an application is scheduled in favor of its preference versus against its preference. The observed performance improvements range from 3.98% to 114.48%. The experiment results indicate that different applications have different scheduling preferences. The performance improvement due to being properly scheduled varies from application to application.

3 SWOMPS: Design and Implementation

3.1 Scheduling Requirements

Learning from the experience introduced in Sect. 2, we believe a scheduler is needed to help improve application performance and the whole system's efficiency. The scheduler should fulfill the following requirements:

- **The scheduler should be system-wide that it could schedule multiple applications running currently**
  General OpenMP applications could not perfectly scale with the increasing of processor cores. When the number of available cores exceeds the requirements of a single OpenMP application, it is necessary to share the computer among multiple applications. A scheduler supporting multiple concurrently running applications is needed.
- **The scheduler should have respect for applications' scheduling preferences and resolve possible preference conflicts**
  Each OpenMP application has its specific scheduling preference. The scheduler should be aware of it and perform scheduling accordingly. When the scheduling preferences of concurrently running applications conflict, the scheduler should resolve the conflicts in favor of specific optimizing goal. For example, one way to optimize the system throughput is to weight each applications by the performance impact of its preferred scheduling, and satisfy the applications in the descending order.
- **Each thread should be bound to a core and this binding should be kept as long as possible**
  Scheduler should bind each thread to a specific core to favor its memory access. However, scheduler might need to reschedule the bindings when a new application comes to run or an application terminates. Rebinding a thread to a new core will need to warm up the cache again and may increase memory accesses to non-local memory. When the scheduler needs to adjust its previous scheduling, it should find a scheduling plan that rebinds previous threads as few as possible.

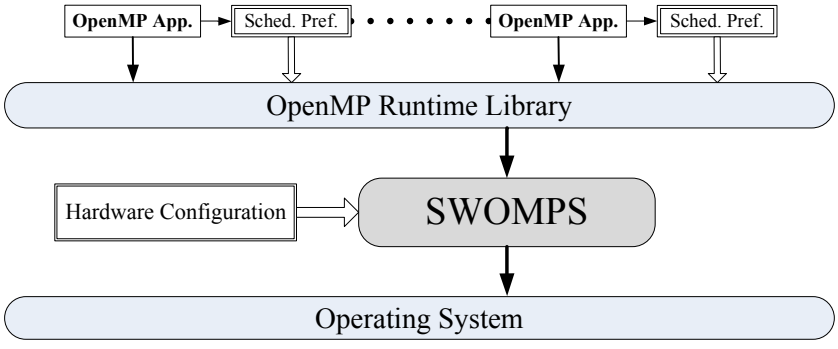


Fig. 2. SWOMPS Architecture

3.2 SWOMPS Work Flow

We have designed a prototype scheduler, SWOMPS (**S**ystem-**W**ide **O**pen**MP** application **S**cheduler) according to the observations introduced previously. Figure 2 shows how SWOMPS cooperates with other parts in the system. And its work flow is as following:

1. SWOMPS starts as a daemon. It inquiries the hardware configuration and initializes internal system model.
2. An application reports its scheduling preference to a database when it starts execution. The scheduling preference is described as an integer. The negative value indicates the application prefers scatter-scheduling, and the positive value indicates the application prefers gather-scheduling. The absolute value of the integer indicates the potential performance impact if the application is properly scheduled. The preference may vary with underlying system, it should not be hard coded. In our prototype implementation, the scheduling preference is stored in an environment variable. Moreover, for many real world applications, the preference may vary due to the alternation of different computational kernels. We use an overall preference for each application for simplicity.
3. OpenMP runtime library inquiries the scheduling preference after creating threads pool. It sends the preference as well as application process ID and thread IDs to SWOMPS.
4. When the application terminates, OpenMP runtime library sends the process ID to SWOMPS.
5. SWOMPS generates a scheduling plan whenever it receives a message. The scheduling plan for each process is calculated according to the current states of the system and the changes of the work load, either a group of threads start execution or they terminate. And the plan is carried out with the help of operating system.

3.3 Scheduling Algorithm

There are two phases in the scheduling algorithm of SWOMPS. The first phase assigns the core quota in each processor to the applications that they can bind their threads

onto, which is described in Algo. 1. The second phase decides the actual thread-core bindings, and it is described in Algo. 2.

Algorithm 1. Assign Core-Quota of Each Processor to Applications

Input: $Pref(app)$: scheduling preference of application app
Input: $PrevCoreQuota(app, p)$: In the previous scheduling, how many threads of application app had been bound to cores in processor p
Result: $CoreQuota(app, p)$: In the new scheduling, how many threads of application app could be bound to cores in processor p

```
1  m ← number of total processor cores;
2  foreach application, app, in descending order of |Pref(app)| do
3      CandidateCnt(app) ← min{m, number of threads in app};
4      m ← m − CandidateCnt(app);
5  foreach processor, p, do AvailCoreCnt(p) ← number of cores in p ;
6  foreach application, app, in descending order of |Pref(app)| do
7      foreach processor, p, do CoreQuota(app, p) ← 0 ;
8      n ← CandidateCnt(app);
9      while n > 0 do
10         p0 ← arg max_p ⟨AvailCoreCnt(p), PrevCoreQuota(app, p)⟩;
11         if Pref(app) > 0 then
12             t ← 1;
13         else
14             t ← min{n, AvailCoreCnt(p0)};
15         AvailCoreCnt(p0) ← AvailCoreCnt(p0) − t;
16         CoreQuota(app, p0) ← CoreQuota(app, p0) + t;
17         n ← n − t;
```

If the number of currently running threads in all applications is larger than the number of processor cores, SWOMPS will firstly satisfy applications with larger $|Pref(app)|$. The rest threads are not bound to any processor core and left to OS for scheduling. This is carried out by assigning $CandidateCnt(app)$ in the first loop (line 2 to 4) in Algo. 1. The loop iterates the applications from larger $|Pref(app)|$ to smaller ones and assigns core quota to each application. Next SWOMPS further assigns the core quota in each processor to the applications to determine how threads of an application spread among the processors. Again this assignment is carried out from larger $|Pref(app)|$ to smaller ones. For a specific application, as is described from line 9 to 17, SWOMPS will first find a processor that has the most cores unassigned. If there are multiple candidates, SWOMPS will choose one that the application has most thread bound to in the previous scheduling. When SWOMPS recalculats the scheduling plan of a process due to starting or terminating of other processes, it uses $PrevCoreQuota(app, p)$ to avoid unnecessary thread migration. Pseudo code in line 10 accomplishes this goal by finding the maximal tuple $\langle AvailCoreCnt(p), PrevCoreQuota(app, p)\rangle$ and return the relative processor p_0. SWOMPS will assign either one core or as many as possible to the application app according to the scheduling preference of the application and repeat until running out the application's core quota.

There are two steps in deciding the thread-core bindings. In the first step, as described from line 3 to 13 in Algo. 2, if an application has its threads bound to some core in the previous scheduling, these threads are bound to the same core as long as the application has core quota in the processor. In the second step, the rest of the threads are bound to cores where the application has core quota.

Algorithm 2. Generate Thread-Core Bindings

Input: $PrevThreadBind(t)$: the core that thread t bound to in the previous scheduling
Input: $CoreQuota(app, p)$: In the new scheduling, how many threads of application app could be bound to cores in processor p
Result: $ThreadBind(t)$: the core that thread t is bound to in the new scheduling

```
1  RestCoreQuota(.,.) ← CoreQuota(.,.);
2  UnboundCores(.) ← Cores(.);
3  foreach application, app, do
4      UnboundThreads(app) ← {threads of app} ;
5      foreach thread t of app do
6          if PrevThreadBind(t) ≠ ∅ then
7              c ← PrevThreadBind(t);
8              p ← processor that core c belongs to;
9              if RestCoreQuota(app, p) > 0 then
10                 RestCoreQuota(app, p) ← RestCoreQuota(app, p) − 1;
11                 UnboundThreads(app) ← UnboundThreads(app) − {t};
12                 UnboundCores(p) ← UnboundCores(p) − {c};
13                 ThreadBind(t) ← c;
14 foreach application, app, do
15     foreach processor, p, do
16         for i ← 1 to RestCoreQuota(app, p) do
17             t ← any element in UnboundThreads(app);
18             c ← any element in UnboundCores(p);
19             UnboundThreads(app) ← UnboundThreads(app) − {t};
20             UnboundCores(p) ← UnboundCores(p) − {c};
21             ThreadBind(t) ← c;
```

4 SWOMPS Evaluation

We have implemented our prototype scheduler, SWOMPS, with Open64 compiler and evaluated it on a SunFire X4440 server. The configuration of the server can be found in Table 3. The server is equipped with four processors. Each processor has six cores. Each core has separated L1 and L2 caches, and the 6 cores share the L3 cache. The four processors are configured in NUMA architecture. Each processor has 12G local memory. The operating system is Red Hat Enterprise Linux Server 5.4. The version of the Linux kernel is 2.6.18. The operating system allocates new page on the node where the task is running. The OpenMP runtime library is the default library used in Open64 of revision 2722.

We use benchmarks in SpecOMP 2001 test suite in our evaluation. Scheduling preference of each benchmark is set to the performance improvement when the application is properly scheduled. For example, as showed in Tab. 2, *324.apsi* prefers gather-scheduling, the performance improvement is 8.26%, so its scheduling preference is set to 8. While *312.swim* prefers scatter-scheduling, the performance improvement is 114.48%, so its scheduling preference is set to -114. Here we assume that users know application's running characteristic well when it is run exclusively and the scheduler can be guided by users.

4.1 Pairwise Execution

We firstly evaluate the scheduler by concurrently running two applications, each application with 12 threads. We sum up the execution time of the two benchmarks and compare it with the same test without SWOMPS' scheduling. We tested every pair of

Table 3. Experimenting Platform Configuration

Number of Sockets	4
Processor	AMD Opteron 8431
Number of Cores	6-core×4
L1 Cache Configuration	64K×6
L2 Cache Configuration	512K×6
L3 Cache Configuration	6M, Shared
Memory Configuration	12G×4

the 11 benchmarks in SpecOMP 2001. Table 4 lists the performance comparison of tests with and without SWOMPS' scheduling. For each cell in the table, the benchmark listed in the head of its row and its column are the benchmarks that are concurrently executed.

Table 4. Performance Comparison of Tests with and without SWOMPS in Pairwise Execution

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	1.24									
314.mgrid	1.27	1.24								
316.applu	1.11	1.15	1.23							
318.galgel	1.15	1.08	1.24	1.04						
320.equake	1.17	1.31	1.36	1.32	1.11					
324.apsi	1.05	1.28	1.46	1.04	1.00	1.13				
326.gafort	1.17	1.11	1.05	1.08	1.07	1.02	1.12			
328.fma3d	1.15	1.11	1.31	1.13	1.09	1.26	1.06	1.19		
330.art	1.28	1.29	1.41	1.03	1.01	1.22	1.13	1.23	1.15	
332.ammp	1.12	1.40	1.13	1.16	1.11	1.13	1.07	1.11	1.04	1.17

Performance improvement can be observed in 54 out of the 55 testings, except the testing of *<324.apsi, 318.galgel>*. The maximum performance improvement is 46.3% (*<324.apsi, 314.mgrid>*). And the average improvement is 16.5%. SWOMPS showed its efficiency for two concurrently running applications.

4.2 Task Queue Simulation

To further evaluate SWOMPS, we test it in a more complicated running circumstance. We simulates a task queue. Benchmarks enter the queue in a random order, and the time interval between two successively entered benchmarks yields to exponential distribution. The expected value of the distribution is set to 360 with the intent of the queue being empty occasionally. There are at most three applications running currently. Each application has a random number of threads, either 6, 8 or 10. The benchmarks are also from SpecOMP 2001.

In addition to compare testing with and without SWOMPS' scheduling we also compare SWOMPS with a non system-wide scheduler implementation. We evaluate whether the lightweight, non system-wide implementation could be an alternative of SWOMPS. In the non system-wide implementation, the scheduler is linked to the application and becomes a part of it. There is no coordination between different applications. The scheduler can only schedule the application it belongs to. It schedules the

application according to the system state when the application starts execution. And that scheduling will not change till the application terminates.

We run each task queue test three times under different scheduling schemas: no scheduling, non system-wide scheduling and SWOMPS scheduling. We totally generated 10 task queue tests. Figure 3 shows the sum of execution time of the 11 benchmarks in each task queue test. Bars labeled with *No Scheduling* are the results of tests without any scheduling. Bars labeled with *Non SysWide* are the results of tests with non system-wide scheduling. And bars labeled with *SWOMPS* are the results of tests with SWOMPS scheduling.

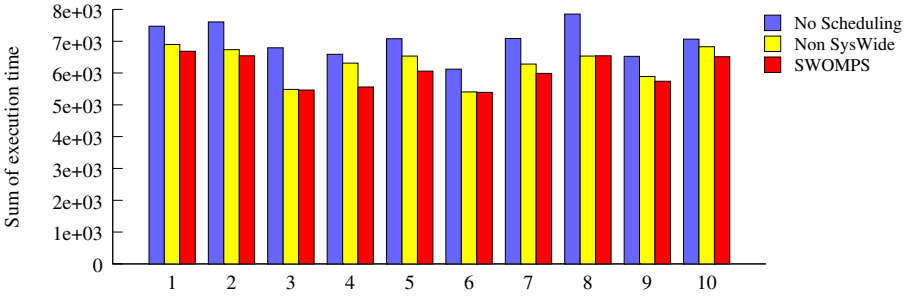


Fig. 3. Sum of Benchmark Execution Time in 10 Random Tests

Both the *Non SysWide* and *SWOMPS* scheduling outperform *No Scheduling* in all the 10 tests. The maximum time reduction is 24%, observed in test 3, for both *Non SysWide* and *SWOMPS*. The minimum time reduction is observed in test 10, 3% for *Non SysWide* and 8% for *SWOMPS*. The average time reduction are 12% for *Non SysWide* and 16% for *SWOMPS*. SWOMPS showed its efficiency in complicated circumstance too.

SWOMPS outperforms *Non SysWide* scheduling. Coordinating concurrently running application system-widely can make better use of hardware resources. Deeper comparisons indicate that system-wide scheduling is necessary in two circumstances, 1) when scheduling preferences of different application conflicts and 2) when an application terminates and more cores are available.

5 Comparison of Exclusive and Concurrent Running Model

In this section, we study the impact of concurrently running multiple applications. There are two concerns in our study, 1) throughput, measured by the reciprocal of the time needed to finish a group of task, and 2) performance of individual application in the concurrent execution.

We firstly enumerate every pair of the 11 benchmarks in SpecOMP 2001 as a task group. The two benchmarks first run exclusively, one after the other. Benchmarks *312.swim, 314.mgrid, 318.galgel* and *320.equake* are executed with 12 threads as their performances are better with 12 threads than with 24 threads. Then the two benchmarks are executed concurrently under SWOMPS' scheduling, each benchmark with

Table 5. Throughput and Performance Study of Pairwise Execution

(a) Throughput comparison between exclusively execution and pairwise execution with 12 threads for each application

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	1.08									
314.mgrid	1.18	1.07								
316.applu	1.01	1.06	1.07							
318.galgel	1.09	1.00	1.18	0.97						
320.equake	1.02	1.13	1.13	1.07	0.91					
324.apsi	1.07	1.25	1.20	1.14	0.92	1.07				
326.gafort	0.99	1.01	1.04	0.96	1.11	0.81	0.94			
328.fma3d	1.22	1.20	1.22	1.15	1.07	1.08	1.11	0.95		
330.art	1.26	1.20	1.24	1.22	1.06	1.17	1.27	0.97	1.31	
332.ammp	1.05	1.06	1.25	0.93	1.27	0.89	0.90	1.14	1.12	1.03

(b) Performance reduction of individual benchmark due to concurrent execution

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	17, 65									
314.mgrid	14, 58	40, 35								
316.applu	30, 54	58, 33	49, 26							
318.galgel	22, 42	45, 14	43, 16	35, 32						
320.equake	28, 36	68, 12	53, 7	52, 28	38, 27					
324.apsi	47, 34	61, 4	52, 3	55, 25	38, 27	45, 22				
326.gafort	39, 54	56, 31	56, 35	45, 41	43, 33	41, 50	32, 50			
328.fma3d	30, 44	57, 21	43, 15	43, 34	32, 27	28, 33	28, 44	44, 44		
330.art	33, 35	63, 8	54, 8	46, 25	35, 23	28, 26	23, 38	35, 39	36, 27	
332.ammp	34, 42	53, 1	52, 6	46, 21	34, 24	34, 36	34, 37	34, 44	34, 41	34, 30

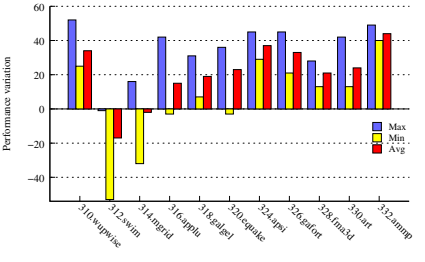
12 threads. We normalize the throughput of the concurrent run with respect to the exclusive run and the results are listed in Table 5a. In 43 of the 55 tests, concurrent run achieves better throughput. The best throughput improvement is 31% and the average throughput improvement is 9%.

We also tested concurrent run without SWOMPS' scheduling. Due to space limit, the detail of the result is not presented. In those tests, none of the concurrent run has better throughput. The average throuughput decreases by 32%. Simply running multiple applications concurrently does not guarantee throughput improvement.

We compare the performance of individual benchmark between exclusive run and concurrent run. The results are listed in Table 5b. For example, "17, 65" in the second row and second column indicates that, compared with exclusive run, when *312.swim*

Benchmark sequence in the task queue	N_T
332, 316, 326, 314, 328, 320, 330, 324, 310, 318, 312	1.56
328, 330, 326, 312, 320, 316, 332, 324, 318, 310, 314	1.50
312, 314, 328, 318, 310, 330, 320, 316, 326, 332, 324	1.54
326, 332, 330, 320, 314, 312, 310, 318, 324, 316, 328	1.49
330, 332, 318, 324, 310, 312, 320, 316, 314, 326, 328	1.49
330, 310, 324, 332, 328, 312, 314, 326, 316, 320, 318	1.35
312, 314, 330, 310, 324, 328, 316, 332, 320, 326, 318	1.39
332, 316, 314, 324, 328, 326, 312, 318, 310, 320, 330	1.54
316, 312, 320, 332, 326, 324, 310, 318, 330, 328, 314	1.50
326, 316, 320, 310, 314, 332, 318, 324, 328, 330, 312	1.56

(a) Throughput comparison



(b) Performance variation

Fig. 4. Throughput and Performance Study of Task Group with 11 Benchmarks

and *310.wupwise* run concurrently, performance of *312.swim* decreases 15%, and performance of *310.wupwise* decreases 65%. In the 110 comparisons, 90 of them are less than 50%, and the average performance decrease is 35.6%. This result is better than expected considering only half of the threads running and the L3 cache and memory bandwidth are shared among two benchmarks.

We next use all the 11 benchmarks as a task group. The 11 benchmarks enter a task queue in a random order. In the exclusive run, benchmarks *312.swim, 314.mgrid, 318.galgel* and *320.equake* are executed with 12 threads, others are executed with 24 threads. At the same time, only one benchmark is running. In the concurrently run, each benchmark runs with 12 threads. At the same time, there are two benchmarks running. When a benchmark terminates, the next benchmark in the task queue starts execution immediately. We tested 10 randomly generated testings and normalize the throughput of the concurrent run with respect to the exclusive run. Figure 4a lists the order of the 11 benchmarks in the queue in the first column. The second column lists the normalized throughput. On average, concurrently run has 49% improvement in throughput.

Figure 4b shows the performance variations of concurrent run compared with exclusive run. Bars labeled with *Max* give the maximum performance decrease of concurrent run in the 10 tests. Bars labeled with *Min* are the minimum performance decrease. Bars labeled with *Avg* are the average performance decrease. A negative value means benchmark runs faster in the concurrent run. Significant performance improvements have been observed in the concurrent run of benchmark *312.swim* and *314.mgrid*. A further study shows the major source of the improvement is thread binding. As we can see form the figure, the average performance decrease of the 11 benchmark are all less than 50%.

These experiments show that concurrent running model outperforms exclusive running model in throughput, especially if there are many tasks. Meanwhile, concurrent running will slow down the applications but they are reasonable.

6 Related Work

Many researches [1,2,3,4,5,6] have been carried out to improve the performance of OpenMP applications on multi-core system. Truong et al. [1] seek to improve scalability from implementation aspect. Their work introduces *thread subteams* to overcome the thread mapping problem and enhance modularity. Noronha et al. [2] study the benefits from using large page support for OpenMP applications. Terboven et al. [4] improve data and thread affinity of OpenMP programs on multi-core system by binding thread to thread cores and allocate memory with *next touch* strategy. A series of papers [3,5,6] from Broquedis et al. introduce a runtime system that transpose affinities of thread teams into scheduling hints. With the help of the introduced *BubbleSched* platform, they propose scheduling strategy suited to irregular and massive nested parallelism over hierarchical architectures. They also propose a NUMA-aware memory management subsystem to facilitate data affinity exploitation.

To the authors' knowledge, there was no study about improving concurrently running multiple OpenMP applications.

7 Conclusion

Restrictions of heating and power consumption has converted the effect of Moore's law from increasing the performance of a single-core processor to the number of cores in a processor. More and more cores will be available in one processor. However, a preliminary experimental study of the scalability of OpenMP applications shows that OpenMP applications cannot efficiently utilize the increasing processor cores in general. Concurrently running multiple OpenMP applications will become a common usage model.

In the NUMA architecture with multi-core processors, the processor cores are asymmetric. How the threads of the concurrently running OpenMP applications are distributed on the processor cores is important to the performance of all the current applications. In this paper, we proposed a system-wide scheduler, SWOMPS, to help schedule the threads on the processor cores. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application. Experiment results shows that SWOMPS is efficient in improving the whole system performance. We also compared the exclusive running and concurrent running with SWOMPS. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual application in concurrent execution is reasonable.

References

1. Chapman, B.M., Huang, L.: Enhancing OpenMP and its implementation for programming multicore systems. In: Bischof, C.H., Bücker, H.M., Gibbon, P., Joubert, G.R., Lippert, T., Mohr, B., Peters, F.J. (eds.) PARCO. Advances in Parallel Computing, vol. 15, pp. 3–18. IOS Press, Amsterdam (2007)
2. Noronha, R., Panda, D.K.: Improving scalability of OpenMP applications on multi-core systems using large page support. In: IPDPS, pp. 1–8. IEEE, Los Alamitos (2007)
3. 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)
4. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: MAW '08: Proceedings of the 2008 workshop on Memory access on future processors, pp. 377–384. ACM, New York (2008)
5. 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)
6. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.A.: Dynamic task and data placement over numa architectures: An OpenMP runtime perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)
7. Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., Boku, T.: Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 15–27. Springer, Heidelberg (2009)

8. Terboven, C., an Mey, D., Sarholz, S.: OpenMP on multicore architectures. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 54–64. Springer, Heidelberg (2008)
9. Curtis-Maury, M., Ding, X., Antonopoulos, C.D., Nikolopoulos, D.S.: An evaluation of OpenMP on current and emerging multithreaded/multicore processors. 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. 133–144. Springer, Heidelberg (2008)

Topology-Aware OpenMP Process Scheduling*

Peter Thoman, Hans Moritsch, and Thomas Fahringer

University of Innsbruck
Distributed and Parallel Systems Group
A6020 Innsbruck, Austria
peter.thoman@uibk.ac.at

Abstract. Multi-core multi-processor machines provide parallelism at multiple levels, including CPUs, cores and hardware multithreading. Elements at each level in this hierarchy potentially exhibit heterogeneous memory access latencies. Due to these issues and the high degree of hardware parallelism, existing OpenMP applications often fail to use the whole system effectively. To increase throughput and decrease power consumption of OpenMP systems employed in HPC settings we propose and implement process-level scheduling of OpenMP parallel regions. We present a number of scheduling optimizations based on system topology information, and evaluate their effectiveness in terms of metrics calculated in simulations as well as experimentally obtained performance and power consumption results. On 32 core machines our methods achieve performance improvements of up to 33% as compared to standard OS-level scheduling, and reduce power consumption by an average of 12% for long-term tests.

1 Introduction

OpenMP [1] is one of the most widely used languages for programming shared memory systems, particularly in the field of High Performance Computing (HPC) [2]. Due to recent developments in hardware manufacturing, the number of cores in shared memory systems is rising sharply. Nowadays it is not unusual to find 16 or more cores in a single multi-socket multi-core system, possibly with an even larger number of hardware threads. The topology of these systems is often complex, with hierarchies comprising multiple levels of cache and heterogeneous access latencies in a non-uniform memory architecture (NUMA). Future many-core architectures [3] are likely to further increase the architectural complexity.

This paper presents experiments demonstrating that many existing OpenMP applications and implementations fail to scale fully on such shared-memory, multiprocess systems (SMMPs). They also do not take into account modern CPU power saving technologies increasingly employed in the name of green computing, which usually work on a per-socket basis [4]. As one possible way to overcome these difficulties and enhance throughput we propose the *centralized process-level*

* The research described in this paper is partially funded by the Tiroler Zukunftsstiftung as part of the "Parallel Computing for Manycore Computers" project.

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 96–108, 2010.

scheduling of multiple OpenMP workloads (*jobs*), taking into account any available topology information. This approach is immediately applicable to existing applications without any code-level changes, a significant advantage considering the large number of OpenMP codes in active HPC use.

We have implemented such a system and evaluated its performance. Our contributions are as follows:

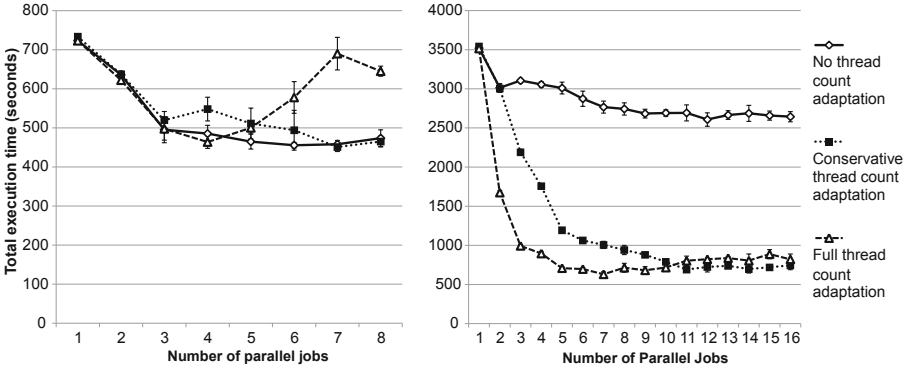
- A client/server architecture for centralized mapping of all OpenMP parallel workloads in a system to the available hardware resources.
- An implementation of this architecture in the Insieme OpenMP compiler and runtime system [5].
- A topology-aware scheduling algorithm optimizing throughput and power consumption of SMMPs processing OpenMP workloads.
- Evaluation and analysis of the actual performance of our architecture and scheduling algorithm in terms of both execution time and power consumption. We compare our results to the Insieme compiler without centralized management as well as to results obtained using GCC's GOMP [6].

The remainder of this paper is structured as follows: In the next section, we present benchmarks and analysis serving to explicate the problem and motivate our approach. Section 3 gathers some references to related work. Section 4 describes the architecture as well as the implementation of our client/server OpenMP runtime system and scheduling algorithm. The results of simulations and experimental evaluation are gathered in Section 5. Finally Section 6 presents a conclusion, and an outlook on potential future improvements.

2 Motivation

We start our discussion by assuming a n-core SMMP system and m OpenMP programs (*jobs*) that should be executed on this system. Additional jobs can be added at any time. This situation corresponds to a realistic scenario in scientific computing and is the basic assumption for the experimental setting adopted throughout this paper. There are a number of natural options for executing multiple OpenMP jobs:

- Sequentially execute the jobs, and have each job allocate n threads – the default specified by the OpenMP standard. Standard queuing system in HPC clusters use this method.
- Start all m jobs in parallel and leave thread scheduling to the OS. As we will show, this option can have a severe detrimental impact on the resulting performance (see Section 5.2).
- Run less than m jobs in parallel, each of them using less than or equal n threads. A standard OpenMP implementation enables this option, does however require some manual queuing. The thread scheduling is still left to the OS.



(a) Set of 8 jobs, large problem sizes. (b) Set of 16 jobs, various problem sizes.

Fig. 1. Initial Experiments

Figure 1 shows some initial benchmarking results. For a complete description of the experimental setup and hardware used throughout this paper see Section 5.2. The three parallel execution strategies shown relate to the options presented above as follows: *No thread count adaptation* simply uses n threads per job ignoring the number of parallel jobs running in the system; *Conservative thread count adaptation* uses $\min(n, \lfloor n/(0.5 * m) \rfloor)$ threads; *Full thread count adaptation* uses $\frac{n}{m}$ threads. With the first option, many more threads are running on the system than hardware cores are available, while for the third option the numbers are equal. The conservative option presents a compromise between these two strategies.

In both Figure 1(a) and 1(b), simple serial execution (1 parallel job) is clearly shown to be far from ideal. For the batch of 8 large jobs featured in the first experiment, an improvement in total runtime of 37% compared to serial can be achieved by exploiting job parallelism, while for the second batch an impressive 5-fold increase in throughput was measured. This underutilized performance potential is the motivation behind our approach of introducing a novel, job-level OpenMP scheduling.

2.1 Scaling Behavior of Popular OpenMP Codes

In order to explain the performance improvements of job-parallel OpenMP execution demonstrated above we investigated the scalability of the OpenMP codes contained in the NAS parallel benchmarks [7] (BT, LU, MG, CG, IS, FT and EP) and two locally developed simple kernels (mmul and gauss, performing dense matrix multiplication and Gaussian elimination respectively). While our initial tests dealt with programs as monolithic units, we now determine the scalability of each OpenMP parallel region separately, since the differences between e.g. initialization code and actual computation or different phases of computation make a per-region analysis more informative and effective.

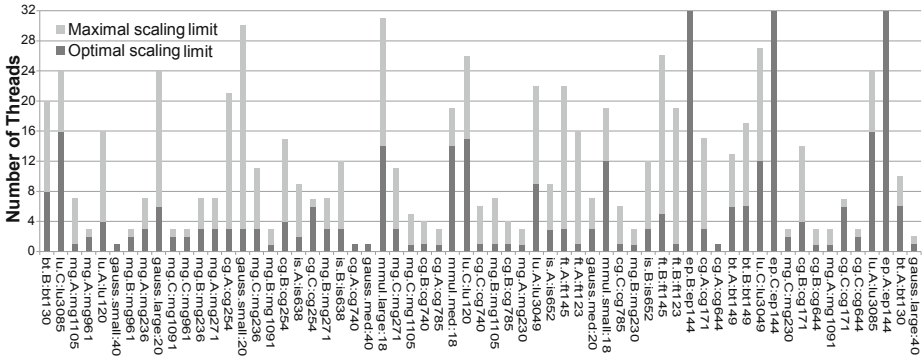


Fig. 2. Per-region scaling of OpenMP codes. Regions are shown along the x-axis.

Figure 2 shows the scaling behavior of each parallel region contained in the test programs. Regions in the chart are identified by the program name, program size and line number of the first statement inside the region. As an example, `bt.B:bt130` identifies the parallel region starting at line 130 of the BT benchmark in the NAS suite, when run at problem size B.

Note that, while this approach provides a relatively fine-grained view of the performance characteristics of the parallel regions, it does not account for the effects of external system load on region scalability. All test were performed on an unloaded system.

Optimal scaling limit denotes the upper limit on the number of threads at which the speedup obtained lies within 20% of ideal. The *maximal scaling limit* is the number of threads with the largest absolute speedup. Using the latter achieves the lowest execution time for a single job, while the former is more power efficient and less wasteful if multiple jobs are to be run in parallel. Note that, for many parallel regions, the default OpenMP behaviour of choosing the same number of threads across all program regions as there are hardware cores available is greatly inefficient on our 32 core target system.

3 Related Work

Enhancing OpenMP to make better use of locality and increase scalability, particularly on multicore architectures, is a topic that has been repeatedly investigated over the past years [8,9]. Recently, particular attention has been paid to scheduling tasks (as introduced by OpenMP 3) [10]. However, these efforts focus on improving the scalability and performance of single OpenMP programs. Conversely, our system overcomes individual program's scalability limitations by scheduling additional programs, optimizing the whole system's throughput. In the existing literature, locality is often signaled by nested parallel regions, which are still not widely used in practice. Our system simply improves locality of subsequent threads of each process, but does so for multiple processes at a

time. A popular approach to developing scalable software for large machines is hybrid OpenMP/MPI programming [11]. Since this method produces multiple OpenMP processes, it is complementary to our process-level scheduling system.

One of the primary goals of multiprocessor scheduling at the OS level is fairness [12], while our OpenMP scheduling system is intended to optimize throughput. However, the defining difference between OS-level scheduling and OpenMP process scheduling is that in the latter case the number of threads used can be determined by the scheduler itself, while it is fixed by user-level applications in the former. Also, when dealing only with OpenMP programs, some more useful assumptions can be made, like data locality being most important between subsequent threads (due to the default loop distribution strategies defined in OpenMP).

4 Architecture

Our topology-aware OpenMP job scheduling system consists of three major components:

- The **Insieme compiler** [5], a source to source compiler supporting OpenMP which also enables per-region profiling (e.g. for OpenMP parallel regions) and unique region identifiers.
- The **Insieme OpenMP runtime library**, which works like a standard OpenMP library for most operations, but communicates with a central server when opening and closing a parallel region.
- The **Insieme OpenMP server**, a management process that keeps track of available CPU cores and outstanding requests for threads, and makes mapping and scheduling decisions based on system topology and load.

4.1 Process Communication

Process communication is required to implement the centralized management of system resources across OpenMP processes. In practice this means that multiple OpenMP programs – upon encountering the start of a parallel region – request hardware resources from the management process (server), including information about the expected scalability of the related region. The server then decides which and how many logical cores to dedicate to the requesting process and sends a reply indicating them.

In our system this communication is achieved by means of UNIX V message queues [13]. The message queue mechanism was chosen because of its good semantic fit with the desired functionality and relatively low overhead of less than 6 microseconds for each paired send/receive operation on our hardware.

Figure 3 illustrates the typical communication operations associated with each parallel region in the source program. *PID* stands for the unique process id of the user program, *optcount* and *maxcount* refer to the scaling descriptors introduced in Section 2.1 and grey arrows represent communication over a message queue.

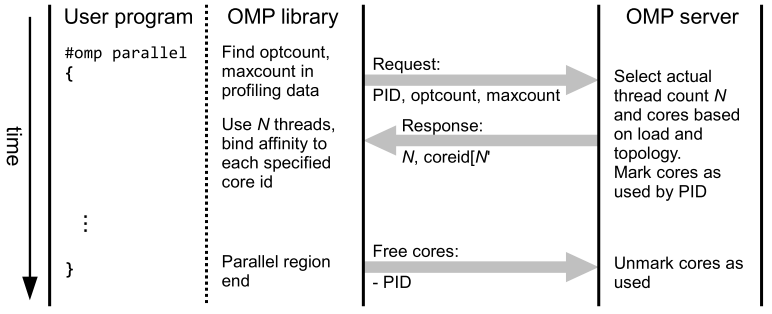


Fig. 3. Process communication

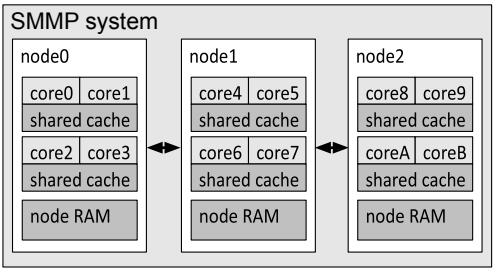
When a new parallel region is encountered, its unique identifier (generated at compile time) is looked up in the table of available profiling information by the runtime system. The retrieved data is included in a request dispatched to the central OpenMP process, which additionally includes the process ID of the user program making the request. Upon receiving this request, the server makes its mapping and scheduling decisions (see next section for details). Unless the system is fully loaded (in which case the request is postponed) a reply is sent immediately, containing the number of threads/cores assigned (N) as well as a list of core ids to be used. The runtime library on the client side then allocates and/or reuses the required number of threads, and binds each of them to a specific core as specified by the list sent by the server. After dispatching the reply the server flags the cores it indicated as in use in its internal data structures.

4.2 Topology-Aware Scheduling

In multi-socket multi-core NUMA systems, there are multiple levels of memory hierarchy to be aware of, with significant differences in terms of access latency and bandwidth. An example of such a hierarchy, ordered from fastest to slowest: L1 cache, L2 cache, shared L3 cache, node local RAM, RAM one hop distant, RAM n hops distant. We call a scheduling process *topology-aware* if it seeks to minimize memory accesses to slow, distant memories by explicitly making use of information on the structure of a system.

One critical difference between scheduling as it is generally encountered in e.g. OS-level schedulers and our OpenMP job scheduling is that, due to the flexibility of most OpenMP programs, we can freely decide not just which threads to run when and on which hardware, but also the number of threads to use for specific regions of a program. This decision should be based on knowledge about the scalability of the regions in question (currently gained through profiling) as well as the current and expected future load of the system.

For the implementation of our system we use information gained from `lib numa` [14] and the Linux `/proc/cpuinfo` mechanism to construct a distance matrix with one distance value for each pair of logical cores. From here on, we refer to the the entry at position i, j in this matrix as $dist(i, j)$. Figure 4(a)



(a)

	0	1	2	3	4	5	6	7	8	9	A	B
0	0	0	1	1	10	10	10	10	20	20	20	20
1	0	0	1	1	10	10	10	10	20	20	20	20
2	1	1	0	0	10	10	10	10	20	20	20	20
3	1	1	0	0	10	10	10	10	20	20	20	20
4	10	10	10	10	0	0	1	1	10	10	10	10
5	10	10	10	10	0	0	1	1	10	10	10	10
6	10	10	10	10	1	1	0	0	10	10	10	10
7	10	10	10	10	1	1	0	0	10	10	10	10
8	20	20	20	20	10	10	10	10	0	0	1	1
9	20	20	20	20	10	10	10	10	0	0	1	1
A	20	20	20	20	10	10	10	10	1	1	0	0
B	20	20	20	20	10	10	10	10	1	1	0	0

(b)

Fig. 4. Topological core distance example

provides an example topology of a relatively small system, and Table 4(b) shows the corresponding distance matrix. Note that the factor of distance amplification for each higher level of hierarchy can be chosen arbitrarily, but should always be larger than the maximum distance possible on the previous level. For clarity we have chosen a value of 10 in the example.

Cores are selected based on a greedy algorithm that locally minimizes the distance from the previously selected core to the next one. This is performed rapidly via lookups in statically cached lists of close cores. While not always resulting in a globally optimal core selection, the low overhead and importance of local distances for many algorithms (see Section 5.1) make this method well suited for our purpose.

Our OpenMP job server supports some flags to adjust the core selection and mapping process, which will be described next. *Fragmentation* in this context means that, over time, threads belonging to many different OpenMP processes will be assigned to cores belonging to a single topological unit (e.g. a node), due to earlier scheduling decisions. This has negative effects on locality and the effectiveness of shared caches.

Locality. Turns on and off the use of topology information to improve locality. Useful to check the base assumption that higher locality improves performance.

Clustering. Reduces fragmentation over longer running periods by preferentially maintaining clusters of close resources as free or occupied. This approach can also reduce power consumption on systems with per-node power management (see Section 5.2). Small overhead costs in processing time and server memory requirements.

Enhanced Clustering. In conjunction with clustering, allows the server to further reduce fragmentation by slightly decreasing the number of cores provided to a process in cases where a new, previously unused set of cores would become fragmented.

Strict Thread Counts. If enabled, the Insieme OpenMP server may postpone requests when highly loaded instead of starting them with a smaller number of threads than ideal. Beneficial when there is a mixture of jobs with varying scalability.

The impact of these options is examined in Section 5. Note that minimizing distance by means of clustered scheduling is not always ideal on a NUMA system which is not fully loaded, since memory bandwidth intensive applications may benefit from threads being spread out across nodes. However, our algorithm is optimized for the case of the full system being utilized by numerous threads from multiple processes.

Figure 5 provides an overview of the decision algorithm performed by the server when receiving a new request for threads from a client process. The *close core list* is a list precomputed for each core that contains the core ids of all other cores in the system, listed in order of topological distance. For clustering, a *core set* is a set of cores which share some level of memory hierarchy. Consequently there can be multiple levels of core sets, in which case the algorithm tries to find suitable cores starting from the lowest level of hierarchy. In the example shown in Fig. 1 there are 2 levels of core sets, the first sharing cache (e.g. cores 0 and 1) and the second sharing memory banks (e.g. cores 0 to 3).

When a client sends a message signaling the end of a parallel region, the cores are marked as free and, if outstanding requests are in the FIFO queue, they are processed as above.

CA	number of cores available
RT	target number of cores
C, PC	core identifiers
LC	list of core identifiers returned
`optcount`, `maxcount`	request parameters (Sec. 4.1)
[`strict`], [`local`], [`clustering`], ...	scheduling flags (Sec. 4.2)
`loadfactor`, `threshold`	$\in [0, 1]$ depending on system state

$RT = \texttt{optcount} + \texttt{loadfactor} * (\texttt{maxcount} - \texttt{optcount})$
if $CA < 0$ **or** ([`strict`] **and** $CA <$ `optcount`) **then**
  put current request on FIFO queue
  return {}
end if
while $RT > 0$ **do**
  C = Free core
  if [`local`] **then**
    Choose C from close core list
    if [`clustering`] **and** C from new set **then**
      Prefer C from (in order):
      ↪ Occupied core set containing exactly RT free cores
      ↪ Occupied core set containing $> RT$ free cores
      ↪ Any free core set
    end if
    if [`enhancedclustering`] **and** $\text{dist}(PC, C) > \texttt{threshold} * |\text{size}(LC) - RT|$ **then**
      return LC
    end if
    $LC = LC \cap \{C\}$
    $PC = C$
    $RT = RT - 1$
  end if
end while
return LC

Fig. 5. Core selection algorithm

5 Evaluation

In this section our system and algorithm are evaluated, first by performing simulations and calculating some theoretical metrics and second by performing experiments and measuring runtimes and power usage. All experiments were performed on Sun X4600 M2 servers with AMD Opteron 8356 processors. This is an 8 socket architecture, with 4 cores each containing private L1 and L2 caches and sharing 2 MB of L3 cache. The sockets have a distance of one to three hops each [15]. The systems run CentOS version 5 (kernel 2.6.18) 64 bits. To compile the reference version of the example programs, GCC version 4.3.3 was used with the -O3 option to reflect a production environment. For our own version, SVN revision 277 of the Insieme source-to-source compiler and runtime system was employed, using the same GCC version and options as above to perform back-end compilation.

To ensure statistical significance each experiment was repeated 10 times, and the median result is reported. In charts vertical error bars are used to show the standard deviation of a set of experiments.

5.1 Simulation

To evaluate the impact of the scheduling options presented in Section 4.2 we performed a simulation of client requests and calculated the following metrics:

Overhead. The average amount of time required to make a scheduling decision, in microseconds.

Target miss rate. The difference between the desired number of threads (RT) and the number actually provided (N).

Three distance metrics. $LC = \{c_1, ..., c_N\}$ denoting the set of cores provided:

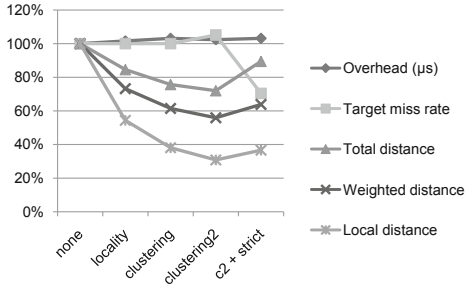
- Total distance: $\sum_{i=1}^{N} \sum_{j=1}^{N} dist(c_i, c_j)$
- Weighted distance: $\sum_{i=1}^{N} \sum_{j=1}^{N} \frac{dist(c_i,c_j)}{|i-j|+1}$
- Local distance: $\sum_{i=1}^{N-1} dist(c_i, c_{i+1})$

Which distance metric has the best predictive qualities depends on the access patterns the code exhibits. Weighted and local distance are more significant than total distance for many real-world OpenMP kernels which rely on the default distribution of loop iterations, resulting in data access locality being higher in subsequent (according to OpenMP numbering) threads.

Figure 6 shows the results of a simulation of 1000 requests to the OpenMP server, with four different configurations corresponding to different settings of the flags introduced in Section 4.2: *none* disables all flags; *locality* enables the use of locality; *clustering* enables locality and clustering; *clustering2* enables locality, clustering and enhanced clustering; *c2 + strict* enables all flags. The simulation uses the same system topology as the experimental setup described in Section 5.2, randomly simulating jobs with sizes normally distributed and ranging from 1 to 32 threads.

			Distance		
	μs	MR	Total	Weight	Local
none	1.26	2.94	8003	2098	893
locality	1.28	2.94	6765	1537	485
clustering	1.3	2.94	6054	1289	340
clustering2	1.29	3.09	5760	1173	275
c2 + strict	1.3	2.07	7159	1340	327

(a)



(b)

Fig. 6. Metrics computed in simulation

Table 6(a) shows the raw values while Fig. 6(b) illustrates the relative impact of the different scheduling options by normalizing the values. The columns contain, in order from left to right: the amount of time, in microseconds, required for the scheduling decision; the target miss rate; and the three distance metrics described above.

The impact on response time and target miss rate of the locality and clustering options is negligible, but they can reduce the weighted distance of the returned set of cores by around 40% and the local distance by up to 64%. Enabling the strict thread counts option significantly reduces the miss rate, but at the cost of reduced locality. In the next section the practical impact of these options will be evaluated.

5.2 Experiments

Figure 7 shows the results of a small-scale experiment performed to compare the total runtime of a fixed set of benchmarks using traditional sequential execution, OS-based parallel execution and our client/server scheduling system with various options. The specific set of programs used was the following (randomly selected from the set of test applications introduced in Section 2.1): *mg.C, gauss.large,*

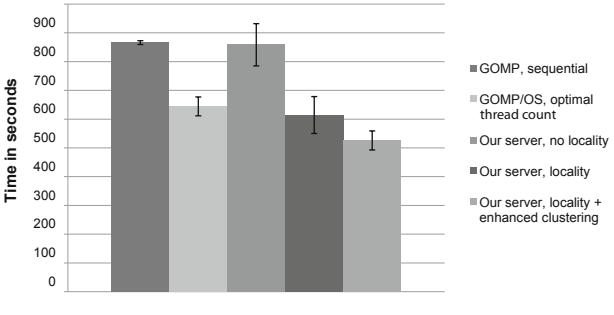


Fig. 7. Small-scale experiment

is.A, matrixmult.medium, cg.A, gauss.small, bt.B, mg.A, ep.B, ft.A, is.A, lu.A, matrixmult.small.

The theoretical advantages of locality-based scheduling and clustering shown previously are confirmed by an improvement of 28% by exploiting the former and 39% by additionally enhancing the latter, compared to using our system without making use of topology information. The improvement compared to traditional sequential execution is 40%, and 19% remain when comparing standard OS parallel scheduling of the processes and statically forcing optimal thread counts. In other words, in this experiment, a reduction in runtime of around 21% can be achieved by improving the exploitation of hardware parallelism by selectively using multiple parallel processes, and an additional 19% gain is possible by enhancing the locality of thread sets via topology-aware scheduling.

Large-scale Experiments and Power Consumption. As a second step of evaluation we performed a large scale experiment. Over 5 hours, a new OpenMP process was randomly selected and started every 10 to 60 seconds. The programs were again chosen from the set introduced in Section 2.1. The random number generator seeds for program and interval selection were kept constant throughout the experiment to guarantee repeatability and comparability.

During the runtime of the experiments, we continuously measured and logged the system power consumption using the Sun ILOM service [15], which allows for a resolution of several measurements per second.

Table 1. Performance results of 5 hour experiment

Scheduling type	Total Time (s)	% of sequential
Sequential	28356	100.00 %
OS parallel, no limit	121855	429.73 %
OS parallel, limit 8	82417	290.65 %
OS parallel, limit 2	23591	83.20 %
server, OS	21527	75.00 %
server, no locality	27959	98.60 %
server, locality	21240	74.91 %
server, clustering	18941	66.80 %

Table 1 lists the total runtime required to finish execution of all the programs launched during the 5 hour testing period, for various scheduling methods. *Sequential* refers to standard sequential execution, *OS parallel* executes a number (up to some limit) of processes in parallel without any explicit scheduling and *Server* uses our system. *Server, OS* only assigns the amount of threads to use, but leaves their placement up to the OS. The other options enable the corresponding flags described in Section 5.1).

Fully parallel execution using standard OS scheduling leads to a very large number of active threads and a performance collapse due to context switching overhead. While this method can achieve good results in the small-scale experiments shown in Section 2 care must be taken to select a suitable limit for production use. In this experiment, a limit of two parallel processes leads to an improvement of 17% compared to sequential execution.

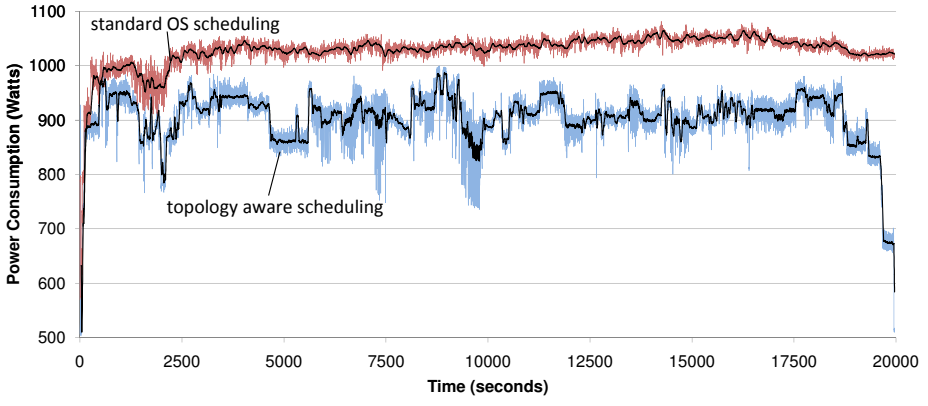


Fig. 8. Power consumption over time during large-scale experiment for standard OS scheduling and topology-aware, clustered scheduling using our system

Our client/server mapping system performs well, greatly improving throughput compared to traditional sequential execution or non-managed parallel processes. With locality information and clustering the best result is achieved, a 33% improvement compared to sequential execution.

However, there is another advantage to offered by clustered thread scheduling and affinity mapping. By reducing inter-node communication and preferentially keeping entire nodes and related core sets empty this technique allows hardware power saving technologies to function more effectively. On our target system, and most servers currently in use, power saving technologies like frequency scaling only work on a per-node basis and not on the individual cores of a node. Clustered, locality-aware scheduling preferentially keeps entire nodes free of work, allowing them to enter an appropriate low-power state.

Figure 8 shows the measured power consumption, over the time period of the experiment, of standard OS scheduling and clustered scheduling using our server. The average power consumption of the former is 1014 Watts, which the latter reduces by 12% to 904 Watts. Note that around 8 measurements per second are taken, and that the black lines represent a central moving average over 25 data points.

6 Conclusion

We presented an OpenMP process-level scheduling and mapping solution that uses system topology information to improve thread locality. Additionally, a suitable number of threads is automatically selected for each OpenMP parallel region depending on scalability estimates and current system load. The implementation comprises a source-to-source compiler, a server process that manages hardware resources and an OpenMP runtime library which communicates with the server, allocates threads and performs affinity mapping as instructed. Simulation and both small- and large-scale benchmarks show consistent improve-

ments in throughput and, with clustering, a marked decrease in average power consumption can be observed.

One drawback of our method we have not addressed yet is the need for reliable per-region scaling data. This necessitates either developer-supplied information or instrumented benchmarking runs. In the future we plan to alleviate this issue using machine learning, with estimated scalability data gained by static analysis during compile time and further refinement during runtime.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface. Version 3.0 (May 2008)
2. Karl-Filip, F. (ed.), Bengtsson, C., Brorsson, M., Grahn, H., Hagersten, E., Jonsson, B., Kessler, C., Lisper, B., Stenström, P., Svensson, B.: Multicore computing – the state of the art (2008), http://eprints.sics.se/3546/
3. 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)
4. Herbert, S., Marculescu, D.: Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In: Proc. 2007 Int. Symp. on Low Power Electronics and Design ISLPED '07, pp. 38–43. ACM, New York (2007)
5. http://www.dps.uibk.ac.at/projects/insieme/
6. Novillo, D.: OpenMP and automatic parallelization in GCC. GCC developers summit (2006)
7. Bailey, D., Barton, J., Lasinski, T., Simon, H.: The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA (1991)
8. Chapman, B., Huang, L.: Enhancing OpenMP and Its Implementation for Programming Multicore Systems. In: Advances in Parallel Computing, vol. 15. IOS Press, Amsterdam (2008)
9. Noronha, R., Panda, D.K.: Improving Scalability of OpenMP Applications on Multi-core Systems Using Large Page Support. In: Parallel and Distributed Processing Symp., IPDPS 2007, March 26-30. IEEE International, Los Alamitos (2007)
10. Duran, A., Corbalan, 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)
11. Krawezik, G., Cappello, F.: Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In: ACM SPAA 2003, San Diego, USA (June 2003)
12. Li, T., Baumberger, D., Hahn, S.: Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. SIGPLAN Not. 44(4), 65–74 (2009)
13. Stevens, W.R.: Advanced Programming in the UNIX Environment. Addison Wesley Longman Publishing Co., Inc., Amsterdam (1992)
14. Kleen, A.: A NUMA API for Linux (2004), http://halobates.de/numaapi3.pdf
15. Sun Microsystems, Inc.: Sun Fire X4600 M2 Server Architecture. White Paper (2008), http://www.sun.com/servers/x64/x4600/arch-wp.pdf

How to Reconcile Event-Based Performance Analysis with Tasking in OpenMP*

Daniel Lorenz[1], Bernd Mohr[1], Christian Rössel[1],
Dirk Schmidl[2], and Felix Wolf[1,2,3]

[1] Forschungszentrum Jülich, Jülich Supercomputing Centre, Germany
[2] RWTH Aachen University, Dept. of Computer Science, Germany
[3] German Research School for Simulation Sciences, Aachen, Germany

Abstract. With version 3.0, the OpenMP specification introduced a task construct and with it an additional dimension of concurrency. While offering a convenient means to express task parallelism, the new construct presents a serious challenge to event-based performance analysis. Since tasking may disrupt the classic sequence of region entry and exit events, essential analysis procedures such as reconstructing dynamic call paths or correctly attributing performance metrics to individual task region instances may become impossible. To overcome this limitation, we describe a portable method to distinguish individual task instances and to track their suspension and resumption with event-based instrumentation. Implemented as an extension of the OPARI source-code instrumenter, our portable solution supports C/C++ programs with tied tasks and with untied tasks that are suspended only at implied scheduling points, while introducing only negligible measurement overhead. Finally, we discuss possible extensions of the OpenMP specification to provide general support for task identifiers with untied tasks.

1 Introduction

In parallel computing, a task denotes an independent unit of work that contributes to the solution of a larger problem. A task is usually specified as a sequence of instructions to process a given subproblem. Tasks can be assigned to different threads and can be executed concurrently with other tasks, as long as input and output dependencies between tasks are observed. To offer a more convenient way of expressing task parallelism, version 3.0 of the OpenMP specification [1] introduced a task construct along with synchronization mechanisms and task scheduling rules.

The OpenMP specification distinguishes between tied and untied tasks. Tied tasks can be suspended only at special scheduling points such as creation, completion, taskwait regions, or barriers. In contrast, untied tasks can be suspended at

* This material is based upon work supported by the US Department of Energy under Grant No. DE-SC0001621 and by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IS07005C.

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 109–121, 2010.

any point. In addition, a tied task can be resumed only by the thread that started its execution, whereas an untied task can be resumed by any thread of the team.

The task construct provides an additional concurrency dimension within OpenMP programs, as threads can migrate between tasks and tasks can migrate between threads, although the latter option is available only for untied tasks. This creates a challenge for traditional event-based performance analysis, which instruments certain code locations such as code region entries and exits. Event-based analysis characterizes the control flow of a thread as a sequence of code region enter and exit events, whose consistency may be disrupted by the task scheduler. For example, tasking may violate the proper nesting of enter and exit events, impeding the reconstruction of dynamic call paths or the distinction between inclusive and exclusive performance metrics for a given code region. Furthermore, the sudden suspension of one task in favor of another makes it hard to correctly attribute performance metrics to the first task.

To monitor the missing events that can restore the consistency of the observed sequence and properly expose this additional dimension of concurrency to performance measurement, we have developed portable methods

- To track task suspension and resumption so that it becomes known when a task region is left and reentered;
- To identify individual task instances so that different instances of the same task construct can be properly distinguished; and
- To recognize parent-child relationships between tasks so that it can be determined whether one task acts on behalf of another.

Implemented as an extension of the automatic source-code instrumenter OPARI [10], our solution supports C/C++ programs with tied tasks and with untied tasks that are suspended only at implied scheduling points, a restriction that some OpenMP implementations (e.g., the current Sun compiler) still satisfy. Introducing only negligible measurement overhead, the extra instrumentation inserted for tasking opens the door to a rich variety of performance-analysis applications including the mapping of task instances onto threads.

The outline of the paper is as follows: We start with a survey of related work in Section 2. Then, we present a detailed problem statement in Section 3. After explaining how to obtain unique task identifiers in Section 4, we describe how to establish parent-child relationships between tasks in Section 5. In Section 6, we discuss possible extensions of the OpenMP specification to provide general support for task identifiers with untied tasks. Automated instrumentation for tracking task suspension and resumption with OPARI is the subject of Section 7, followed by an experimental evaluation with respect to measurement dilation in Section 8. The last section draws a conclusion and outlines future work.

2 Related Work

Unlike MPI, OpenMP does not specify a standard way of monitoring the dynamic behavior of OpenMP programs, although in the past various proposals have been presented.

The first proposal for a portable OpenMP monitoring interface (named POMP) was written by Mohr et. al [10]. Based on an abstract OpenMP execution model, POMP specifies the names and properties of a set of callback functions, including where and when they are invoked. The proposal also describes the reference implementation OPARI, a portable source-to-source translation tool that inserts POMP calls in Fortran, C, and C++ programs. OPARI is widely used for OpenMP instrumentation, for example, in the performance tools Scalasca [6] and TAU[11]. In an attempt to standardize an OpenMP monitoring interface, a second and significantly enhanced version of POMP [9] was developed by a larger group of people, taking into account experiences with OPARI, the European INTONE project, and the GuideView performance tool from KAI. A prototype tool based on binary instrumentation for this second version of POMP was implemented by DeRose et al. [3]. However, the OpenMP ARB decided to reject POMP 2 because it was seen as too complex and costly to implement and also dropped the idea of standardizing an official monitoring interface for OpenMP. The tools subgroup of the OpenMP ARB agreed on publishing a whitepaper describing a much simpler and less powerful monitoring interface based on a proposal by Sun [7]. OpenMP compiler vendors are encouraged to follow this specification if they want to provide tool support for their OpenMP implementations. To our knowledge, only Sun and Intel include (undocumented and incomplete) implementations of this interface in their compiler offerings. Finally, the group around the OpenUH research compiler investigated various ways of compiler-based instrumentation for OpenMP monitoring [2].

All this work was carried out before the introduction of tasks in OpenMP. Very little has been done so far on tools supporting the monitoring of OpenMP tasks. Fürlinger et al. [5] did initial work on instrumenting OpenMP tasks using OPARI. However, their solution, which simply encloses task regions with enter end exit calls, supports only tied tasks and lacks a task-identification mechanism. In addition, Lin and Mazurov [8] extended the whitepaper API to support tasking and presented a prototype implementation based on the Sun Studio Performance Analyzer.

3 An Additional Concurrency Dimension

Before version 3.0, an OpenMP program had just one concurrency dimension – threads. The latest version of the OpenMP specification added a second dimension. Similar to a thread, a task can be created, suspended, resumed, and carries some state with it. This can have a serious impact on traditional event-based performance analysis tools, which usually consider only the first concurrency dimension.

Event-based performance analysis tools, which are also sometimes referred to as direct-instrumentation tools, instrument certain points in the code, usually the entries and exits of nested code regions, and trigger an event whenever such a point is reached. At runtime, the execution of a thread then appears as a sequence of events delineating nested code region instances. For each code region instance, performance metrics can be individually calculated. Code region instances executed

```
1  #pragma omp task // Task 1
2  {
3    f1();
4  }
5  #pragma omp task // Task 2
6  {
7    f2();
8  }
9  #pragma omp taskwait
10
11 void f1()
12 {
13   enter_event(f1);
14   #pragma omp task // Task 3
15   {
16     // do something
17   }
18   #pragma omp taskwait
19   exit_event(f1);
20 }
21
22 void f2()
23 {
24   enter_event(f2);
25   #pragma omp task // Task 4
26   {
27     // do something else
28   }
29   #pragma omp taskwait
30   exit_event(f2);
31 }
```

Fig. 1. Example OpenMP code with tasking

within one another are assumed to be executed on behalf of the enclosing code region instance, establishing the notion of inclusive and exclusive performance metrics (i.e., covering or not covering child region instances, respectively).

Tasking now disrupts this event sequence. A task can be suspended in the middle, and another arbitrary task can be assigned to the executing thread instead. As a consequence, a thread may switch between tasks potentially without any parent-child relationship between them. However, since the OpenMP specification provides no method for identifying such relationships, it becomes very hard to calculate meaningful inclusive metrics. Moreover, the execution of tasks may not be properly nested so that call paths can no longer be calculated in the traditional way. Even worse, defining call paths along the execution of tasks without a parent-child relationship might not make sense at all.

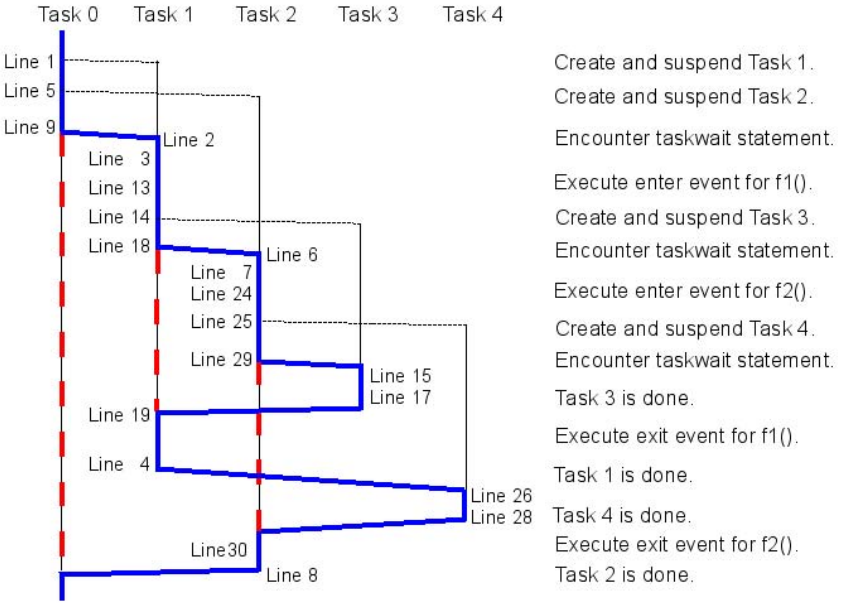


Fig. 2. Timelines of tasks in the program from Figure 1 using a FIFO scheduler. Each task is represented as a separate timeline. The continuous thick line indicates the task currently being executed. The dashed thick lines indicate intervals during which tasks are temporarily blocked. The numbers next to the continuous thick line correspond to the source-line numbers in Figure 1.

This problem is illustrated by the example shown in Figure 1. A potential scheduling order for this program when executed with a single thread is given in Figure 2. The example assumes FIFO scheduling with tasks being immediately suspended once they have been created. The code is supposed to be part of an initial Task 0. Obviously, the order of enter and exit events in this scenario violates the nesting property. Although executed by only a single thread, the events appear in the following order:

$$\text{enter f1}() \rightarrow \text{enter f2}() \rightarrow \text{exit f1}() \rightarrow \text{exit f2}()$$

In addition, we can observe that one task (Task 2) was suspended in favor of another (Task 3) that was not a descendant of the previous one. It is clear that the familiar notions of call paths and of inclusive and exclusive metric values cannot be maintained in this type of event sequence.

If task identifiers were available and events were produced on every task switch, we could correctly measure values which are exclusive for each single task and consider call-path information separately for each task instance. However, a complex computation often contains many tasks, which potentially create

child tasks. Having aggregate (inclusive) metrics for a computation including the values of its child tasks eases the location of problem candidates. For this reason, knowing the creation relationships is of importance as well.

4 Obtaining Task Identifiers for Tied Tasks

One way to provide unique task identifiers for tied tasks is to have a *task-private* and globally accessible variable that is uniquely initialized on task creation and remains unchanged and valid throughout the active lifetime of the task – whether functions are called from within the task or whether the task is interrupted during its lifetime.

Such a task-private variable can be emulated using a threadprivate global variable that is constant during the execution of a task but changes its value at task scheduling points. If we proceed with a newly created task, we set the variable to the identifier of the new task, and if we resume a previously suspended task, we set the variable to the corresponding previous value. For this purpose, we need a mechanism to store and to restore task identifiers at scheduling points as well as a method to obtain unique identifiers for new tasks.

This can be implemented by declaring a threadprivate variable `current_task_id` in the global scope that will give us the valid task identifier any time during program execution (see Figure 3). It is initialized with a value corresponding to the initial, implicit task. A function to obtain fresh task identifiers, `get_new_task_id()`, can be implemented by concatenating the 32 bit OpenMP thread identifier with a 32 bit threadprivate variable that is incremented everytime the function is called. The resulting 64-bit combination will provide globally unique identifiers within every parallel region. Another advantage of this combination is that we do not require any synchronization to obtain new and unique task identifiers.

To store and to restore task identifiers, we place a local (i.e., automatic storage) variable `old_task_id` before each scheduling point and initialize it with the value of `current_task_id`. If a task is suspended and resumed later on, the corresponding local variable is still valid and used to restore the value of `current_task_id`.

To maintain task identifiers throughout the entire execution of an application, instrumentation must be applied

- To parallel regions,
- To taskwait as well as implicit and explicit barrier constructs, and
- To task constructs.

Before we enter a parallel region (see Figure 4), we need to store the current task identifier so that it can be restored afterwards. Therefore we add a local variable `old_task_id` and assign to it the identifier of the current task. After the completion of the parallel region, the local variable is still valid and we can restore the original value by assigning `old_task_id` to `current_task_id`.

```
int64_t current_task_id = ROOT_TASK_ID;
#pragma omp threadprivate(current_task_id)
```

Fig. 3. Declaration and initialization of task identifiers

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp parallel
  {
    current_task_id = get_new_task_id();
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 4. Maintaining task identifiers at parallel regions

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp taskwait
  current_task_id = old_task_id;
}
```

Fig. 5. Storing and resetting task identifiers at taskwait statements (applies also to barrier constructs)

Inside the parallel region, each thread creates an implicit task and we need to obtain a unique identifier for each of these tasks. This is done in parallel by assigning the return values of calls to `get_new_task_id()` to the threadprivate variables `current_task_id` right at the beginning of the parallel region. Each thread/task has now a unique `current_task_id` variable that is valid until we reach the next scheduling point.

Scheduling points within parallel regions occur at `taskwait` and `barrier` as well as at `task` constructs. Here we must maintain task identifiers because the scheduler can suspend the current task and continue either with a newly created or a resumed task. At `taskwait` and `barrier` constructs, we store the `current_task_id` in a new local variable `old_task_id` immediately before reaching the scheduling point and restore it afterwards, as shown in Figure 5. At task creation points the situation is slightly different. In addition to storing and restoring the identifier of the current task, we need to obtain a new task identifier by calling `get_new_task_id()` and assign it to `current_task_id`, similar to the procedure used for parallel regions. This is demonstrated in Figure 6.

If applied to all parallel regions and scheduling points, the code sequences in Figure 4, Figure 5 and Figure 6 are sufficient to maintain task identifiers throughout the entire execution of a program.

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp task
  {
    current_task_id = get_new_task_id();
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 6. Maintaining task identifiers at task creation points

5 Tracking the Task Creation Hierarchy

With task identifiers available, a task creation hierarchy can be constructed. The instrumentation presented in the previous section allows the identifier of the parent task to be obtained at the task creation point. Subsequently, the direct parent-child relationship can be easily extended to a full pedigree by appending the new task as a child node of the parent task node to the tree of the creation hierarchy.

```
{
  int64_t old_task_id = current_task_id;
  #pragma omp task firstprivate(old_task_id)
  {
    current_task_id = get_new_task_id();
    add_child(current_task_id, old_task_id);
    // do something
  }
  current_task_id = old_task_id;
}
```

Fig. 7. Tracking the task creation hierarchy

When a new task is created, the parent task must have stored its identifier in the local variable `old_task_id` in order to restore the identifier when continuing its execution. Making the value of `old_task_id` firstprivate in the child task ensures that it is initialized with the parent's identifier, establishing a connection between the two. The instrumentation for task creation is shown in Figure 7, assuming the creation tree is built using a function named `add_child()`.

6 Untied Tasks

The mechanisms described so far assume that scheduling happens only at implied scheduling points that can be easily instrumented. This assumption is always true for tied tasks. Compared to tied tasks, the scheduling of untied tasks is more flexible in two ways. An untied task

– May be resumed by a thread different from the one that executed the task before it was suspended and
– May be suspended at any point, not only at implied scheduling points.

The first condition does not cause any problem as long as rescheduling occurs only at scheduling points. Since a performance-analysis tool using our interface is able to verify the identity of a thread, it can easily distinguish between different threads. For this reason, our solution also works with OpenMP implementations that reschedule untied tasks only at implied scheduling points. Although not prescribed by the OpenMP specification, some implementations (e.g., the current Sun compiler) still follow this rule because rescheduling at scheduling points, where control is trivially transferred to the OpenMP runtime, is technically simpler than it is at arbitrary points. However, untied tasks that are rescheduled at arbitrary points may disrupt the whole measurement.

General support for untied tasks requires additional services from the runtime system. At least, notification on task scheduling events is necessary so that whenever a task is preempted performance metrics can be collected and the task identifier can be set to the new task. One way of implementing such a notification mechanism would be the option to register a callback function `cb_resume_task()` with the OpenMP runtime that is called whenever the execution of a task is started or resumed. Inside `cb_resume_task()`, the environment of the task brought to execution should be visible. Furthermore, because the runtime system must maintain task identifiers anyway, it would be helpful and probably more efficient if the OpenMP runtime system offered a standard way of obtaining task identifiers rather than having to maintain them on the user level. While in our view an extension of the OpenMP specification would be the ideal solution, the current situation leaves us with only the following options:

– Exploit the fact that some OpenMP implementations (e.g., Sun) suspend untied tasks only at scheduling points.
– Let the instrumentation make all tasks tied. This changes the behavior of the measured program, but in some cases still allows a few very limited conclusions to be drawn (e.g., on the granularity of tasks).

7 Automated Instrumentation with OPARI

OPARI [10] is a source-to-source instrumentation tool for OpenMP programs. To allow automated instrumentation of OpenMP C/C++ programs with tasking, OPARI was extended to instrument also the task and taskwait constructs. Furthermore, the instrumentation of OpenMP directives that contain (implicit) scheduling points or create implicit tasks was modified to maintain and to expose task identifiers and parent-child relationships based on the ideas presented in Sections 4 and 5.

Access to `current_task_id` is encapsulated and provided via two functions: `POMP_Get_current_task()` and `POMP_Set_current_task()`. The function

Table 1. Examples of how OPARI instruments tasking-related constructs

OMP directive	instrumented directive

`#pragma omp task`
{
// do something
}
	{
	`POMP_Task_create_begin(pomp_region_1);`
	`POMP_Task_handle pomp_old_task =`
	`POMP_Get_current_task();`
	`#pragma omp task firstprivate(pomp_old_task)`
	{
	`POMP_Set_current_task(POMP_Task_begin(`
	`pomp_old_task, pomp_region_1));`
	{
	// do something
	}
	`POMP_Task_end(pomp_region_1);`
	}
	`POMP_Set_current_task(pomp_old_task);`
	`POMP_Task_create_end(pomp_region_1);`
	}

`#pragma omp taskwait`	{
	`POMP_Taskwait_begin(pomp_region_1);`
	`POMP_Task_handle pomp_old_task =`
	`POMP_Get_current_task();`
	`#pragma omp taskwait`
	`POMP_Set_current_task(pomp_old_task);`
	`POMP_Taskwait_end(pomp_region_1);`
	}

`get_new_task_id()` is not called directly, but inside the region-begin function (e.g., in `POMP_Task_begin()`). Some examples of the instrumentation are shown in Table 1.

8 Overhead

To evaluate the runtime dilation of our instrumentation, we performed two tests, the first one based on an artificial benchmark, the second one based on a realistic code example, the Flexible Image Retrieval Engine (FIRE) code [4]. The instrumented calls generated unique identifiers for each task, but did not measure any further metrics.

8.1 Artificial Benchmark

Our benchmark program contained a parallel region in which 10,000,000 tasks per thread were created. Every task just incremented an integer by one. Before executing the program with four threads, it was instrumented using the extended version of OPARI. The execution time was measured and compared against

Table 2. Comparison of the instrumented against the uninstrumented version of the FIRE code

	runtime		overhead	
threads	not instrumented	instrumented	in %	in seconds
1	522.57 s	527.56 s	0.96 %	4.99 s
2	259.55 s	262.64 s	1.19 %	3.09 s
4	129.52 s	129.93 s	0.32 %	0.41 s
6	86.42 s	86.43 s	0.01 %	0.01 s
8	64.86 s	65.13 s	0.41 %	0.27 s
12	43.13 s	43.00 s	-0.30 %	-0.13 s
16	32.12 s	32.43 s	0.95 %	0.31 s

the uninstrumented version. This test was run on an i686 Linux system with a 2.66 GHz quadcore processor using four threads.

Running the uninstrumented program took 1.89 s, while running the instrumented program took 2.50 s. The difference was 0.61 s or 32.3 %. Ignoring program initialization and the increment instruction inside the tasks, the uninstrumented benchmark spent its execution time almost exclusively managing the tasks. This implies that our instrumentation adds approximately 32.3 % of the task management time to the overall runtime. An absolute overhead of 0.61 s for an application with 10,000,000 tasks per thread doing real work is probably negligible. However, acquisition of performance metrics upon the occurrence of task scheduling events might incur additional overhead.

8.2 The FIRE Code

The Flexible Image Retrieval Engine (FIRE) [4] was developed at the Human Language Technology and Pattern Recognition Group of RWTH Aachen University. The benchmark version subject to our study consists of more than 35,000 lines of C++ code. Given a query image and the number of desired matches k, a score is calculated for every image in the database, and the k database entries with the highest scores are returned. Shared-memory parallelization is obviously more suitable than distributed-memory parallelization for the image retrieval task, as the image database can be easily accessed by all threads and need not be distributed.

The initial parallelization of the FIRE code used nested OpenMP threads on two nesting levels [12]. This version was later modified to use OpenMP tasks instead of nested threads. The task-based version creates one task per query image and inside these tasks every comparison of a query picture with a database entry is represented by another task. This approach offers more flexibility than using nested threads because every thread can work on any task. With nested threads, in contrast, we had to assign a fixed number of threads to the lower nesting level.

For our experiments, we used 18 query images and a database with 1000 elements. Since every comparison generates a task, 18000 tasks were created in total. We ran the code on an IBM eServer LS42 equipped with four AMD Opteron 8356 (Barcelona) processors. We conducted ten test runs with and without instrumentation, while varying the number of threads. The average runtimes are shown in Table 2.

The results clearly show that the overhead generated by the instrumentation is insignificant. The total absolute overhead when one thread is used is about 5 s. Compared to the total runtime of 527.56 s this is less than 1 %. When more threads are used, the overhead scales well with the number of threads. When 16 threads are used the overhead amounts to 0.31 s which is still less than 1 % of the total runtime of 32.43 s. So even for larger numbers of threads, the instrumentation overhead is very low compared to the overall execution time.

9 Conclusion and Future Work

A portable method was presented that allows the execution and scheduling of tied and untied OpenMP tasks to be tracked and exposed to performance measurement. Our method can be applied as long as task scheduling occurs only at the implied scheduling points defined in the OpenMP specification, which is always the case for tied tasks and in some implementations (e.g., the current Sun compiler) even for untied tasks.

Implemented as an extension of OPARI, the necessary instrumentation can be automatically inserted into the source code of OpenMP programs written in C/C++. In a next step, we plan to extend our solution to Fortran. The runtime dilation caused by the instrumentation was shown to be negligible, although we believe that the overhead could probably be further reduced if the task identifier was provided by the runtime environment. Performance tools using OPARI are now encouraged to implement a rich variety of analyses of the events delivered by our interface, taking advantage of the two concurrency dimensions (i.e., threads and tasks) being fully exposed – including the mapping between them. Application candidates include analyzing the task synchronization overhead in view of many small tasks, determining the granularity distribution among tasks, studying the task creation hierarchy, and drawing execution timelines of parallel tasks. Code studies will have to show which ones are most relevant. Displaying data related to the two concurrency dimensions in a meaningful way and handling the potential non-determinism of task scheduling will pose major challenges.

General support for untied tasks, which are in principle allowed to be suspended at arbitrary points, cannot be provided unless the OpenMP runtime exposes task scheduling events, which would require an extension of the OpenMP specification. At a minimum, the user should be given the option to register a function to be called whenever a task's execution is started or resumed.

References

1. OpenMP Architecture Review Board. OpenMP application progam interface version 3.0. Technical report, OpenMP Architecture Review Board (May 2008)
2. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Tafti, D., Gopalkrishnan, P.: Towards an implementation of the OpenMP collector API. In: Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the ParCo 2007 Conference, Jülich, Germany (September 2007)
3. DeRose, L.A., Mohr, B., Seelam, S.R.: Profiling and tracing OpenMP applications with POMP based monitoring libraries. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 47–54. Springer, Heidelberg (2004)
4. Deselaers, T., Keysers, D., Ney, H.: Features for image retrieval - a quantitative comparison. In: Rasmussen, C.E., Bülthoff, H.H., Schölkopf, B., Giese, M.A. (eds.) DAGM 2004. LNCS, vol. 3175, pp. 228–236. Springer, Heidelberg (2004)
5. Führlinger, K., Skinner, D.: Performance profiling for OpenMP tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 132–139. Springer, Heidelberg (2009)
6. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
7. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling. Technical report, Sun Microsystems, Inc. (2007)
8. Lin, Y., Mazurov, O.: Providing observability for OpenMP 3.0 applications. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 104–117. Springer, Heidelberg (2009)
9. Mohr, B., Malony, A.D., Hoppe, H.-C., Schlimbach, F., Haab, G., Hoeflinger, J., Shah, S.: A performance monitoring interface for OpenMP. In: Proceedings of the 4th European Workshop on OpenMP (EWOMP'02), Rome, Italy (September 2002)
10. Mohr, B., Malony, A.D., Shende, S.S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The Journal of Supercomputing 23(1), 105–128 (2002)
11. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–331 (2006)
12. Terboven, C., Deselaers, T., Bischof, C., Ney, H.: Shared-memory parallelization for content-based image retrieval. In: ECCV 2006 Workshop on Computation Intensive Methods for Computer Vision (CIMCV), Graz, Austria (May 2006)

Fuzzy Application Parallelization Using OpenMP

Chantana Chantrapornchai (Phongpensri) and J. Pipatpaisan

Dept. of Computing, Faculty of Science,
Silpakorn University, Thailand, 73000
ctana@su.ac.th

Abstract. Developing fuzzy applications contain many steps. Each part may require lots of computation cycles depending on applications and target platforms. In this work, we study the parallelism in fuzzy systems using openMP and its possibility in embedded plaforms. Two versions of the parallelization are mentioned: fine-grained and coarse-grained parallelism. In our study, we found that the coarse-grained approach is more effective due to the overhead of openMP which becomes more visible in the low-speed CPU. Thus, the coarse-grained approach is suggested. Two versions using paralle-for and section are proposed. Two versions give different speedup rate depending on characteristics of the applications and fuzzy parameters. In general, the experiments convey that as the system runs continuously the openMP implementation can achieve a certain speedup, overcoming the openMP overhead by the proposed parallelization schemes.

Keywords: Fuzzy Applications, Parallel Computing, OpenMP.

1 Introduction

Fuzzy systems are now being used in many consumer electronic devices such as air conditions, washing machines, refrigerators etc. To perform fuzzy computation, several phases are needed. The whole computational time depends on how complications such a system is. For a small simple control system, the computation time is not significant. However, most of cases, the computation is complex, i.e., there exists several controls within one system. Also, among these computations, parallelism is often inherent. Thus, to make the computation effective, parallel implementation is necessary.

Since one of the common use of the fuzzy systems is in embedded devices, the system must be implemented in a hardware. One may choose to implement using a general-purpose processor, fuzzy processor, programmable devices etc. Most of these implementations often consider the implicit parallelism in the applications by using developing several parallel hardware units or employing pipeline processing. Programming these devices often use C or VHDL for convenience. To implement a fuzzy processor or specific application circuit which performs typical parallel fuzzy computations, VHDL may be a choice. For general-purposed CPU, the microcontroller is a choice. However, the microcontroller usually has limitations on memory and CPU speed. Also, it is usually used for simple control applications. To

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 122–132, 2010.

facilitate the complex and parallel controls, multiple microcontrollers may be needed. Implementation on the microcontrollers is often based on C language.

In this paper, we are considering the parallelization approaches of fuzzy systems based on OpenMP. We focus on the embedded platforms particularly FPGA where it can handle complex systems. We expect that our implementation can be mapped to embedded platforms using a multicore, each of which may be a low-cost CPU. The hardware implementation can be based on FPGA. We study two versions of the parallel approaches. It is shown that both of the implementations can achieve the certain speedup when the systems run continuously. The studies also imply the overhead of openMP and the application nature which limit the speedup rate.

Many previous work studied the implementation of openMP on the hardware especially FPGA. Compilers and the optimization for openMP in FPGA were studied[1,2,5]. Extensions to openMP to allow the hardware specification are proposed in [6,10]. Kanaujia et. al. presented an approach to simulate the multicore platform[11]. Leoew et. al. showed the hardware generation from openMP pragma to a synthesizable VHDL[8].

Several work has been done in fuzzy control hardware. Most fuzzy processors often have limitation such as 2 inputs and 1 output rules [6], the shape of the membership is triangular or trapezoid. Some work is based on analogue systems [12]. Some requires extra hardware supports and special instruction sets [13]. Many works are based on VLSI systems such as [7]. A general purpose CPU is another choice which is flexible but may be too much powerful. Microcontrollers often provide a moderate solution since it is programmable and easy to create a prototype. However, many computations in fuzzy systems are expensive, such as the use of floating point, multiplication and division etc. They are not suitable to the 8-bit microcontroller unless certain optimization is done. The example is the commercial work of fuzzyTECH which is based on MCS51 and MCS96[14].

2 Backgrounds

In typical fuzzy systems, a given input is read and then fuzzified to be a fuzzy set. This is called the fuzzification process. After that, the value is given to the inference engine to find out which rules are fired. For each rule that is fired, the corresponding output linguistic variable is marked. This step is called a fuzzy inference process. Then, all the fired output linguistic values are concluded to be a crisp value which is the actual output. This is called the defuzzification process. The output is given to the feedback function which is a computation of some linear/nonlinear function and the result becomes the input again.

Based on these steps, necessary parameters that one needs to come up with when the system is designed are the input and output variables, linguistic variables for each input and output and their membership functions , fuzzy rules, inference method and defuzzification methods.

In the following, we briefly explain steps to establish a fuzzy system according to the above components [9].

1. Inputs and outputs: First designers need to define the number of inputs and output. The universes of each input and output are defined. This defines the domain for each fuzzy set.

2. Linguistic variables and membership functions: For each input and output, one needs to define the set of linguistic variables. Each linguistic variable corresponds to a membership function which specifies a mapping from an element to a degree of membership value ranged [0,1]. The membership function is typically defined by triangular shape, trapezoidal shape, bell shape, etc.

3. Rule set: From the given inputs and output and linguistic variables for each one, the set of rules are defined. Typically, the rules are defined from all possible combinations of input linguistic variables. After that, the rule optimization may be done to minimize the number of rules, the number of inputs for each rule, and to minimize the memory usage for the rule set. Most fuzzy systems require 2 inputs and one output. Many fuzzy hardware implements the 2-dimensional associative memory for the rule set.

4. Inference method: Designers need to specify the fuzzy inference method used to compute the output membership degree. Many operators are defined in the literature [9]. The common one is max-min or max-product. The minimum value between two membership values of the two inputs are used as the cut to the output linguistic variable for each rule. Then the fuzzy set for each output linguistic variable is unioned to become a final set and the final set is defuzzified. For max-product, rather using min operation, the product operation is used.

5. Defuzzification method: After all the inferences are done, the final set is defuzzified to get a crisp output value y*. Several methods can be used to defuzzify such as using the max value (means of max, smallest of max,largest of max), computing a centriod value, using the approaches such as weighted-average, center of sum.

3 Parallelization Using OpenMP

There are various ways of viewing parallelization in such applications. Typically, we divide into fine-grained and coarse-grained parallelisms. For the fine-grained one, in each step mentioned above, it is the parallelization that is implementation in each function. For example, the membership function mapping can be parallelized using for *parallel for* and similarly for the defuzzification method. For the rule inference, each *Max-min* computation is parallelizable using *parallel section*. The similar approach is considered in developing parallel fuzzy inference hardware.

For coarse-grained parallelization, based on the above five steps in the previous sections, one may consider to pipeline the steps. This is done as software pipelining or in pipelined fuzzy processors.

In this work, we target to implement the systems in embedded hardwares which has limited speed. From previous experiments, we found that openMP has certain overhead. If we consider to parallelize in fine-grained levels, the overheads are invincible when the code is ported in the hardware. The overheads are also studied in the experiments. Thus, we consider the coarse-grained parallelization.

The typical fuzzy computation is shown in Figure 1. In this figure, inputs are given the fuzzy processing. The fuzzy processing contains steps as mentioned in Section 2. In particular, there may be serveral fuzzy controls inside a fuzzy processing. Each one may control different outputs. After the output is given, the new inputs are read again. The new inputs may be affected by the outputs and some other parameters. This is a closed-loop control. Normally, the loop is repeated forever.

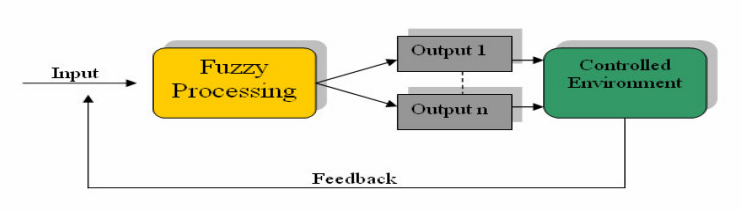


Fig. 1. Original Multiple-output fuzzy controls

3.1 Parallel-Section Approach

The first approach is based on openMP "section" concept. This is depicted in Figure 2. Each fuzzy output control is handled by a thread. At the end of the iteration, the threads are joined together. Then the main thread is ready to compute for the next iteration. This is because the new inputs are computed from the outputs from the previous iteration. Note that dividing the job this way requires the number of outputs be equal to the number threads. Also, the work for each thread may not be the same since each thread performs each fuzzy control which is differed in terms of linguistic variables, rules, etc.

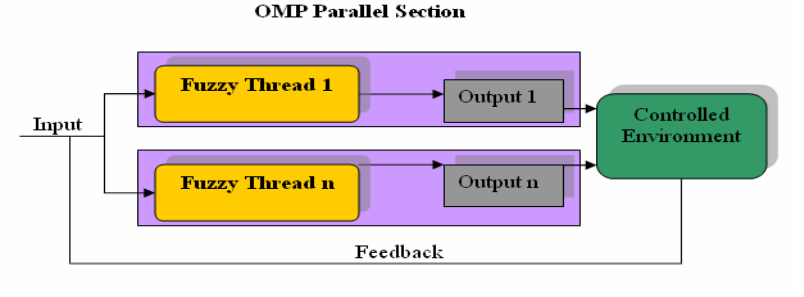


Fig. 2. Parallelization using section

In Figure 2, each subsystem (by the purple region in Figure 2) is surrounded by pragma section as shown in Figure 3(a). For thread mapping, each thread is mapped to each section. Then, each core is assigned to each thread. If the computation time of each thread is not the same, we can see that some threads would be faster and some would be slower. The synchronization overheads are incurred by the blue region in Figure 3(b).

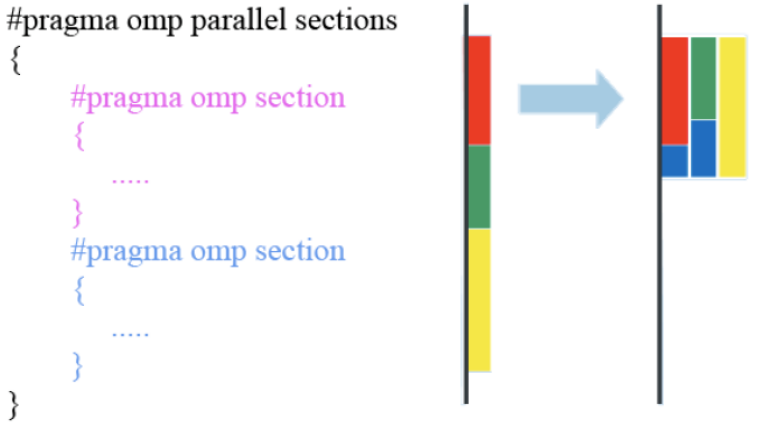


Fig. 3. (a) Section pragma (b) Section distribution to threads

3.2 Parallel-for Approach

Figure 4 shows the parallelization using openMP "for". The purple region shows each loop body. For this kind, the loop is distributed to each thread. Inside the loop, all fuzzy controls are performed. Using this way, we do not require that the number of outputs be the same as the number of threads like in the above approach. Also, the workload per thread is likely to be equal.

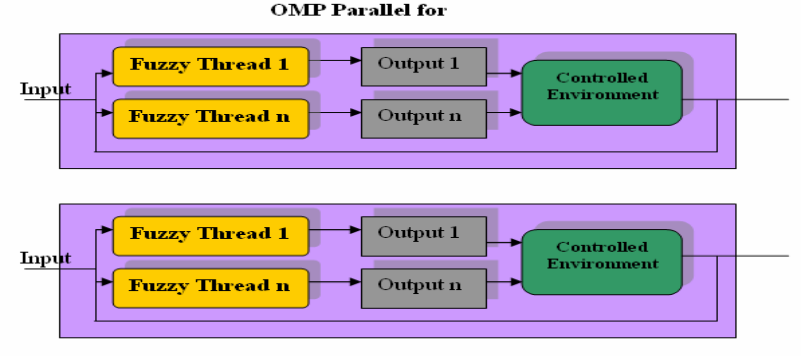


Fig. 4. Parallelization using parallel for

Figure 5(a) shows the parallel-for example. Inside the for loop, we perform all fuzzy system computations. Figure 5(b) shows the distribution for each loop for each thread. However, in the usual control, there is a dependency between each control iteration. The new inputs are usually based on the outputs that control the systems. Thus, in reality, this approach may not be possible.

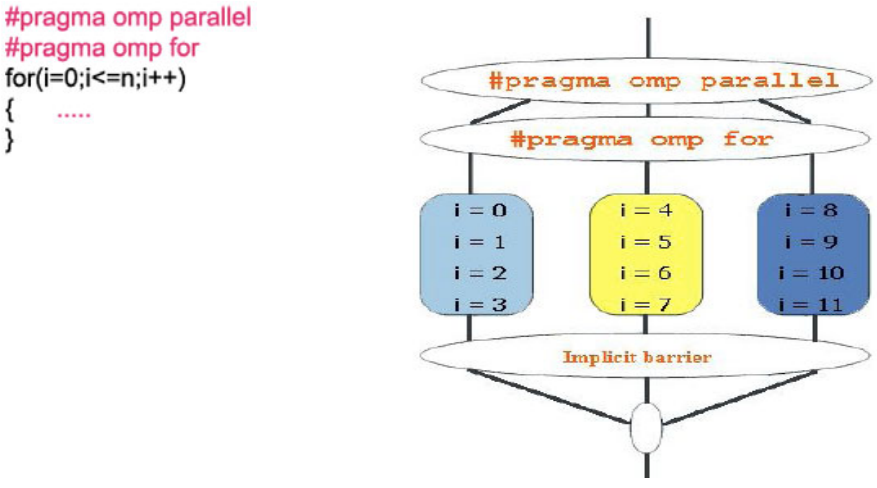


Fig 5. (a) Pragma parallel for (b) work load distribution

In the experiments, we measure the performance for each approach in terms of the speedup gained as we run the system longer.

4 Experiments

In the experiments, we would like to measure the number of cycles used for both parallel versions. We also measure would like to know openMP overhead in order to exlore how the speedup is gained for each type of fuzzy applications. The results will confirm that the parallel implementation using openMP will give benefits as well when implemented in embedded platforms.

Table 1. Example characteristics

Example	# input	#output	#mem func.	#ele per set	#rules	Infer. method	Defuzz.
1.Air condition[2]	3	4	21	51	96	Max-min	Centroid
2.Automatic focusing system[16]	3	3	51	51	27	Max-min	Centroid
3.Reactor temp. controller [16]	3	2	31	21	254	Max-min	Centroid
4. Truck parking [17]	4	2	27	73	130	Max-prod	Fuzzymean

We measure the number of cycles used for each approach. Also, we compared the number of cycles to the serial one. The overheads of openMP pragma are also measured. The experiments are run on the machine with Intel Core 2 Quad, Q8200,

2.33GHz, Core 45nm FSB 1333MHz, L2 Cache 4MB,4GB RAM, running WinXP service pack 3. We use Intel OpenMP/C++ compiler.

In the experiments, we try the examples: air condition control [1], automatic focusing system [16], reactor temperature control [16], and truck parking control [17]. These systems contain more than one output and a number of rules. The characteristics of these examples are shown in Table 1. Column "#input" shows the number of inputs for each system. Column "#output" shows the number of outputs for each system. Columns "#mem func" and "rules" display the number of total membership functions and rules for each system. Columns "Infer. methods" and "Defuzz" display the inference method used and defuzzification method in the example respectively.

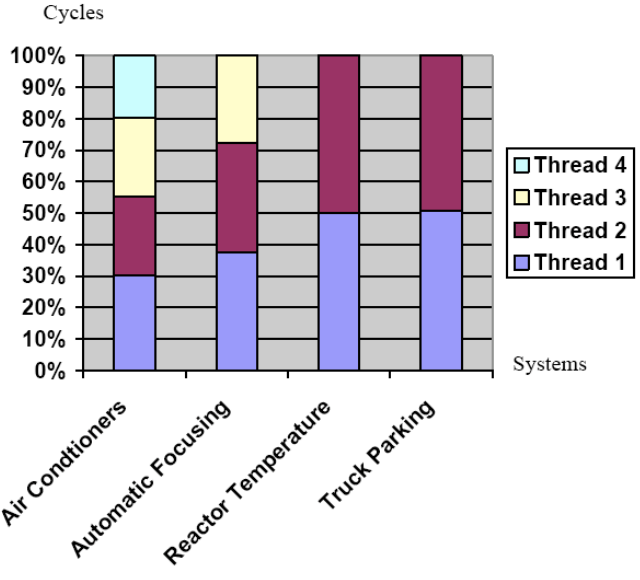


Fig. 5. Work load in percent for each thread

Figure 5 shows the average number of cycles used for each thread in the parallel approaches. In the experiments, we use the number of threads equal to the number of outputs for both versions. The number of threads are not more than the number of available cores in the experiments. We assume that each thread has work about the same size.

Figures 6-7 shows the comparison of the cycles used for the parallel version and serial version for "omp section" and for "omp parallel for" for air conditioner and reactor controller respectively. Figure 6(a) displays the cycles used for omp section and Figure 6(b) shows the number of cycles for omp parallel for respectively. For both examples, we can see that the openMP has fixed overheads. This overhead can be hidden if the systems run long enough. The number of iterations runs to overcome the overheads may be different depending on the complications of each system. For example, for Figure 6, we can see that using the parallel section, it is better than the

parallel-for version. The overheads can be hidden faster. In Figure 7, for both approaches, at least 200 iterations, are needed for hiding the overhead of openMP. However, this measurement is subjective to the CPU clock speed as well. The results only show that with the system running continuously, we can certainly gain benefits from parallelization using openMP without considering the platform.

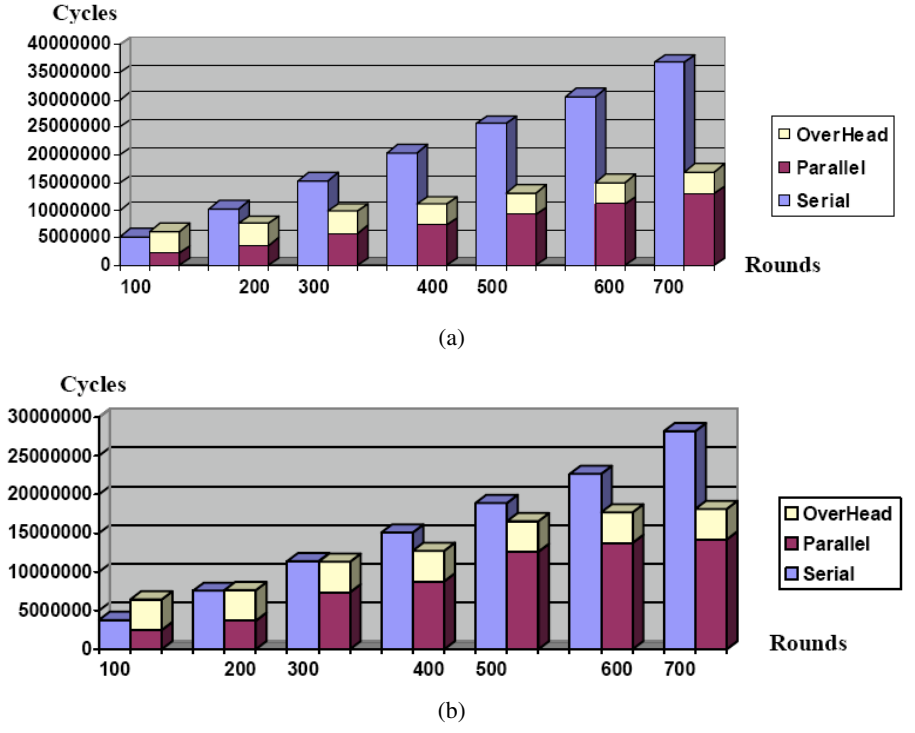
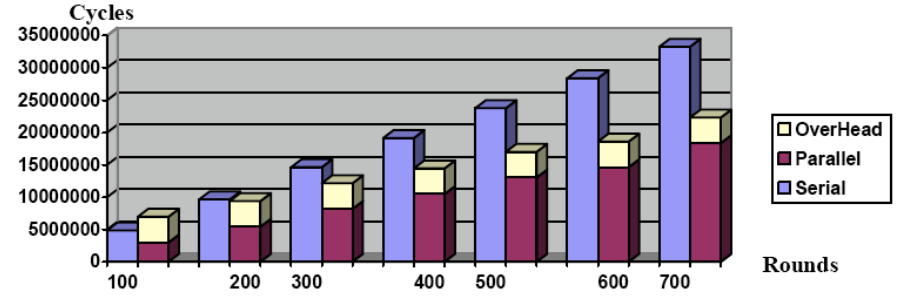


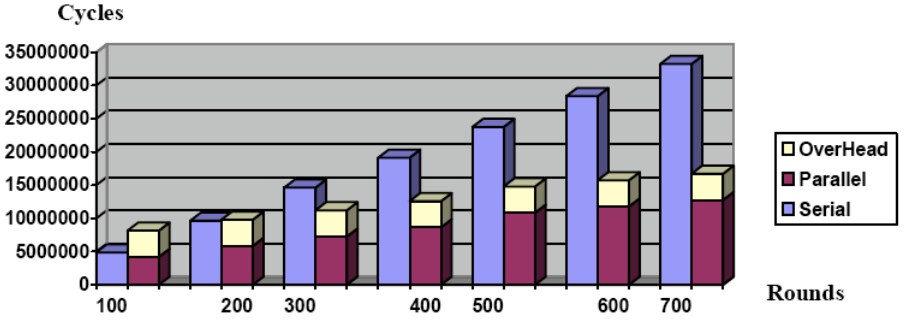
Fig. 6. The cycle measured for each approach for the air condition application (4 threads) (a) omp section (b) omp for

Compared to the fine-grained implementation, for example, each for loop in the defuzzfication may be parallel. The overhead of openMP becomes significant. It also may be magnified when considering the low-speed embedded platform) compared to the parallel speedup gained. The fine-grained approach with openMP is not suitable for these applications.

Figure 8 shows the speedup for both cases. We can see that the parallel-for version obtain better speedup rate that the section version in overall. In air conditioner example, we can reach the speedup to 2.48 while the section version gives speedup 2.18 when running 700 iterations. For truck parking example, we obtain only 1.25 times since there are dependencies in output and input controls. For reactor control and automatic focusing, the increase in the speedup is potential when more number of iterations are run.



(a)

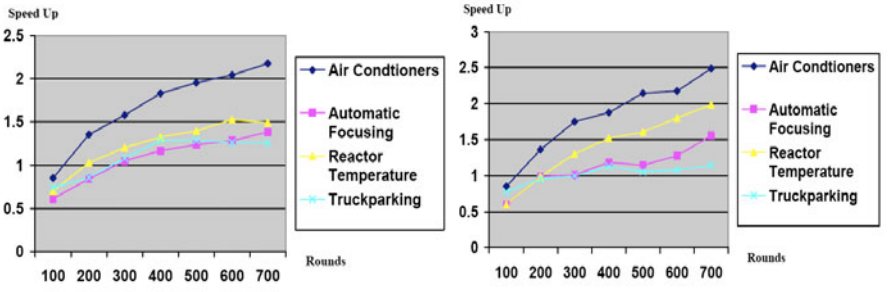


(b)

Fig. 7. The cycles measured for each approach for the reactor controller (2 threads) (a) omp section (b) omp for

In summary, the computation time spent for each thread depends on the type of application and fuzzy computation. From our experiments in Figure 9, we found that most of the intensive computations lie on the inference engine (Rules and Doinfer_Getmax which performs max-min inference and output summarization respectively). Thus, if the system has the large number of rules, it spends more time on the computation. Thus, for each subsystem, if the number of rules is about the same, the workload per thread is about the same for the section approach. Otherwise, the parallel-for approach may be a choice.

We need to consider the overhead of openMP. Thus, the longer the iterations is, the more the overhead can be hidden. The speedup rate depends on factors such as dependency between each fuzzy control, and the sharing of data among the threads. If each fuzzy control can run independently, we would get an ideal speedup. Nevertheless, most of the cases, each fuzzy control shares some data such as membership arrays and possibly rules. Also, some control output may be inputs of another control. This will slow down the speedup. The control dependency case is exhibited in the truck parking application. We can see that the speedup is lower than others.



(a) (b)

Fig. 8. Speedup measured for each approach (a) omp section (b) omp for

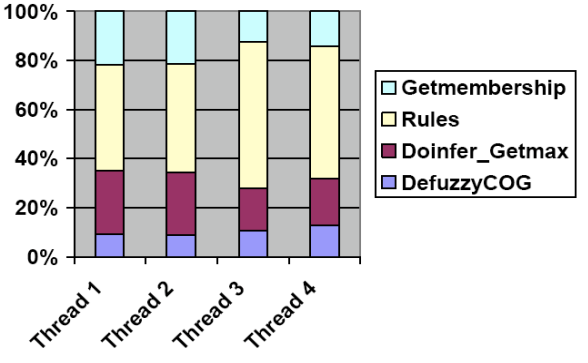


Fig. 9. Average computation for each fuzzy computation steps for Air conditioner

5 Conclusion

In this work, we study various parallelization of fuzzy computation using openMP. The paper mentioned the fine-grained and coarse-grained approaches. The fine-grained approach is not suitable since the overhead of openMP is large when scaled to the low-speed CPU. Thus, the coarse-grained approach is suggested here. Two methods are shown by using "omp section" and "omp parallel for". The investigation shows that using the parallel for approach can be more flexible for various number of threads. However, the dependency between iterations may prohibit the iteration parallelization. OpenMP threads may be considered as another choice for the application. Finally, these threads can be mapped to the FPGA using various implementations such as openUH and others. The mapping to FPGA will be studied further.

Acknowledgement

This work is supported in part by NECTEC (NSTDA) and Faculty of Science funding, Silpakorn University, Thailand.

References

1. Addison, C.: OpenMP 3.0 Tasking Implementation in OpenUH. In: Workshop in Open64 workshop in conjunction with OCG'09 (2009)
2. Amiya, P.: Fuzzy Logic Control of Air Conditioners, http://www.cybergeeks.co.in/projects/projects/Fuzzy%20Logic%20Control%20of%20Air%20Conditioners.pdf (accessed January 5, 2009)
3. Chandrasekaran, S., Hernandez, O., Maskell, D., Chapman, B., Bui, V.: Compilation and Parallelization Techniques with Tool Support to Realize Sequence Alignment Algorithm on FPGA and Multicore. In: Proc. Workshop on New Horizons in Compilers, India (2007)
4. Chapman, B., et al.: Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In: Proc. of Workshop on Multithreaded Architectures and Applications (MTAAP'09) In conjunction with IPDPS 2009, Rome, Italy, May 25-29, pp. 1–8 (2009)
5. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E.: OpenMP extensions for FPGA Accelerators. In: International Symposium on Systems, Architectures, Modeling, and Simulation (2009)
6. Falchieri, D., Gabrielli, A., Gandolfi, E.: A digital fuzzy processor for fuzzy-rule-based systems. Hardware implementation of intelligent systems, 147–164
7. Gabrielli, A., Gandolfi, E., Masetti, M.: Design of a family of VLSI high speed fuzzy processors. In: IEEE Fuzz'96, New Orleans, September 8-11 (1996)
8. Leow, Y.Y., Ng, C.Y., Wong, W.F.: Generating hardware from OpenMP programs. In: IEEE International Conference on Field Programmable Technology (FPT 2006), pp. 73–80 (2006)
9. Ross, T.J.: Fuzzy Sets, Fuzzy Logic and Fuzzy Systems: Theory and Applications. McGraw Hill, New York (1995)
10. Sima, V.-M., Panainte, E.M., Bertels, K.: Resource allocation algorithm and OpenMP extensions for parallel execution on a heterogeneous reconfigurable platform. In: International Conference on Field Programmable Logic and Applications 2008 (FPL 2008), pp. 651–654 (2008)
11. Kanaujia, S., Papazian, I.E., Chamberlain, J., Baxter, J.: FastMP: A Multi-core Simulation Methodology. In: Workshop on Modeling, Benchmarking and Simulation (2006)
12. Song, C.T.P., Quigley, S.F., Pammu, S.: Novel analogue fuzzy inference processor. In: IEEE International Symposium for Circuits and Systems, pp. 247–250
13. Tsutomu, M.: Fuzzy processor, European Patent EP0392494 (1990)
14. http://www.fuzzytech.com
15. http://www.micrium.com
16. http://www.aptronix.com/fuzzynet
17. http://www.imse-cnm.csic.es/Xfuzzy

Hybrid Parallel Programming on SMP Clusters Using XPFortran and OpenMP

Yuanyuan Zhang, Hidetoshi Iwashita, Kuninori Ishii,
Masanori Kaneko, Tomotake Nakamura, and Kohichiro Hotta

Software Development Division,
Next Generation Technical Computing Unit,
Fujitsu Limited,
211-8588 Kawasaki, Japan
{zhang.yuanyuan,iwashita.hideto,ishii.kuninori,
mkaneko,tonakamura,hotta}@jp.fujitsu.com

Abstract. Process-thread hybrid programming paradigm is commonly employed in SMP clusters. XPFortran, a parallel programming language that specifies a set of compiler directives and library routines, can be used to realize process-level parallelism in distributed memory systems. In this paper, we introduce hybrid parallel programming by XPFortran to SMP clusters, in which thread-level parallelism is realized by OpenMP. We present the language support and compiler implementation of OpenMP directives in XPFortran, and show some of our experiences in XPFortran-OpenMP hybrid programming. For nested loops parallelized by process-thread hybrid programming, it's common sense to use process parallelization for outer loops and thread parallelization for inner ones. However, we have found that in some cases it's possible to write XPFortran-OpenMP hybrid program in a reverse way, i.e., OpenMP outside, XPFortran inside. Our evaluation results show that this programming style sometimes delivers better performance than the traditional one. We therefore recommend using the hybrid parallelization flexibly.

1 Introduction

Most supercomputer systems currently functioning as High-Performance Computing(HPC) platforms are SMP(Symmetric Multi-Processing) clusters which are typically characterized with a hierarchical architecture. Each node is typically a shared memory SMP machine made up of multiple cores and these nodes are connected via high-speed network with the inter-node architecture of distributed memory. To derive best performance from such clusters, users must fully make use of such characteristics when writing parallel programs running on such machines. Parallelization among nodes is usually realized by process-level parallelism while that within one node can be realized by process-level or thread-level parallelism, resulting in pure process-level parallelism and hybrid parallelism respectively. Compared to process-level parallelism, thread-level parallelism is said to have the advantages of avoiding communication overhead and

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 133–148, 2010.

reducing memory consumption, and thus to often result in better performance in shared-memory systems in many cases. However, because of the problems such as cache conflict and false sharing in thread-level parallelism, process-level parallelism can sometimes deliver better performance than thread-level one in shared memory systems. Which programming paradigm is better depends on the characteristics of application. In this paper, we focus on process-thread hybrid programming in which thread-level parallelism is implemented by OpenMP, and study how to easily write parallel programs with satisfying performance. Many libraries and programming languages have been proposed to realize process-level parallelism, such as MPI library, High Performance Fortran(HPF), Co-Array Fortran(CAF), and Fujitsu's proprietary language XPFortran. Here we have a short overview on these approaches.

MPI[4] and OpenMP[5] are the current de-facto standards for process-level and thread-level parallelism respectively, and many users use pure MPI or MPI-OpenMP hybrid programming in writing parallel application programs to run on SMP clusters[9]. However, MPI program is said to have some flaws, such as explicit communication, error-proneness, and the difficulty of reading, writing and debugging.

HPF, a high-level data-parallel programming language, adopts a global name-space methodology. Consequently, users write programs from a global view. All variables are located in the global space and users are responsible for specifying how to distribute them on processors. To judge whether a loop can be parallelized or not and in which way to parallelize is directed by user by using HPF directives or left to compiler. Whether communication is necessary or not and the implementation details are left to compiler.

In [10] hybrid programming using HPF is introduced. In this paper the specification of HPF is extended so that the data to be stored in shared memory and accessed by multiple threads in a process are declared by users through extended HPF directives. The compiler translates such declarations to OpenMP directives.

CAF[1] is also an extension of Fortran. It can be used for both process and thread parallelization, and its core features have been included in Fortran 2008 standard. Unlike HPF, CAF defines a local name-space and users manage explicitly locality, data and computation distribution. For process parallelization, each process accesses off-process data explicitly by using an extension of Fortran syntax.

Different from data-parallel languages, in CAF fine grain communication between processes might happen frequently inside computational loops. Because of this, though it is possible to combine CAF and OpenMP to write hybrid programs, there might be frequent communication within thread parallelization, and performance of a CAF-OpenMP hybrid program depends on the communication library used and architecture.

XPFortran, whose predecessor is called VPP Fortran[7], is a data-parallel programming language for process-level parallelism. Like HPF and CAF, XPFortran is also an extension of Fortran. It uses global name-space model and the tedious

low-level details of translating from the global name space to the local ones are left to the compiler. XPFortran defines both global and local memory space. Compared with CAF, in XPFortran users declare data in the local memory space from a global view, that is, before the data are distributed to processors. Different from HPF, in XPFortran users can declare both loop parallelization and communication explicitly. Compared with MPI, the directive programming style of XPFortran makes it easy for users to write parallel programs based on a sequential one step by step, and reduces the code-rewriting cost. On SMP clusters we hope to make hybrid programming easier with XPFortran. Compared to the method introduced in [10], hybrid programming in XPFortran is implemented by realizing thread parallelism in automatic parallelization or permitting users to describe OpenMP directives in XPFortran programs directly. In this paper, we introduce hybrid programming using XPFortran and OpenMP, explain how it's implemented in XPFortran compiler, and show some of our experiences on how XPFortran-OpenMP hybrid programming can be used. Since XPFortran is a data-parallel language similar to HPF, we believe our results here can also be used to HPF and its successors.

The rest of this paper is organized as follows. In Sec. 2, we introduce XPFortran and its compiler implementation. In Sec. 3, we discuss hybrid programming using XPFortran and OpenMP, and the implementation of XPFortran compiler to support such hybrid programming. Sec. 4 presents our performance evaluation results and Sec. 5 concludes the paper.

2 XPFortran Parallel Programming Language

XPFortran is a data-parallel language based on Fortran. It defines some compiler directives and library routines to specify process-level parallelism in Fortran programs. Similar to OpenMP, XPFortran directives can be treated as comments by Fortran compiler and programs written in XPFortran can usually be compiled and executed sequentially. An XPFortran program can also be executed in parallel by multiple processes in Linux or Solaris operating system when it's compiled by XPFortran compiler.

A processing unit for parallel execution of XPFortran programs is called a processor. A collection of processors is called a processor group, organized as an array. XPFortran defines directives called `SPREAD DO` and `SPREAD REGION` for data and task parallelism respectively. A communication directive, the `SPREAD MOVE` directive, is used to transfer data between virtual global memory and local memory introduced below. Directives for synchronization and mutually exclusive execution among processors are also defined.

2.1 Memory Model

As shown in Fig. 1, XPFortran defines both local memory address space and virtual global address space. Local memory is the memory each processor physically owns. It's specific to each processor and therefore always fast to access.

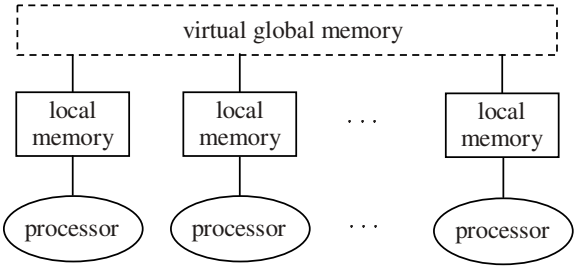


Fig. 1. Memory model of XPFortran

Virtual global memory consists of the total of physical memory. All processors can access data in the global memory, at the cost of communication implemented by compiler implicitly or declared by users explicitly.

Every variable has either a global or local location attribute according to which memory it is located in. Global variables are located in the global memory and shared by all the processors. Local variables can be duplicated or partitioned. As its name hints, duplicated local variable is duplicated in all processors, and reference to such variables results in access to the local one a processor owns. A partitioned local variable is distributed in the specified processors, and each processor can access only the part it owns. By default a variable is a duplicated local variable.

Figure 2 shows some examples to declare global and local variables.

```
      INTEGER AG(100,100), AL(100,100), BL(100,100)
!XOCL PROCESSOR P(4)    ! declare a 1-D processor group with 4 processors
!XOCL GLOBAL AG         ! declare a global array
!XOCL LOCAL AL(/(P),:)  ! declare a local array distributed in 1st-D
!XOCL LOCAL BL          ! declare a duplicated local array
```

Fig. 2. Examples of declaration of global and local variables

2.2 Execution Process

XPFortran uses SPMD(Single Program Multiple Data) execution model. Multiple processes start redundant execution of an XPFortran program from the beginning. `SPREAD DO` directive is used for parallel execution of computational loops. Each iteration of a loop is called a region and these regions are executed in parallel. `SPREAD MOVE` directive describes batch data transfer between global and local memory spaces. Similar to `SPREAD DO` directive, it is followed by a loop block, too. However, data transfer is done only once rather than for each loop iteration. In other words, a unit of processing in `SPREAD DO` construct is an iteration in the loop, and, while it's the whole loop block in `SPREAD MOVE` construct.

For the global and local variables declared in Fig. 2, the following code illustrates example of an XPFortran program. In this example, the outer loop in the loop block immediately following `SPREAD DO` directive is divided and executed by 4 processors in processor group `P` in parallel. For the loop block which follows `SPREAD MOVE` directive, batch communication which transfers partitioned local array `AL` to global array `AG` is executed between processors in `P`.

```
!XOCL SPREAD DO /(P)
      DO I=1,100
         DO J=1,100
            AL(I,J)=I*J
         END DO
      END DO
!XOCL END SPREAD
!XOCL SPREAD MOVE /(P)
      DO I=1,100
         DO J=1,100
            AG(I,J)=AL(I,J)
         END DO
      END DO
!XOCL END SPREAD
```

Fig. 3. Example of some XPFortran directives

2.3 Compilation and Execution of XPFortran Programs

We are developing a compiler for XPFortran, which is a source-to-source compiler. It translates an XPFortran program to a standard Fortran program with calls to MPI library routines. The generated program is compiled by a Fortran compiler and linked with other Fortran and MPI programs if necessary. The machine code generated by Fortran compiler is then executed on target hardware. This process is shown below in Fig. 4. The target hardware of XPFortran is a distributed memory parallel system in which distributed memory nodes are connected and communicate with each other by high-speed interconnect. Each node is typically an SMP machine.

XPFortran compiler transforms global view XPFortran program to local view Fortran-MPI program which describes the actions of each processor. It can be divided into phases as shown in Fig. 5. In the parsing phase, XPFortran source program is converted to intermediate code, on which all operations before code generation are performed. Since the compiler supports the use of the OpenMP directives in XPFortran programs, there are two phases called OpenMP pre-processing and post-processing for the processing of OpenMP directives. The OpenMP pre-processing phase executes syntax analysis, grammar checking, and some pre-processing for OpenMP directives. The OpenMP post-processing phase executes some post-processing operations for OpenMP directives. The details are described in section 3.2. The normalization and optimization phases implement some code optimization. Details of the normalization phase can be found in [6].

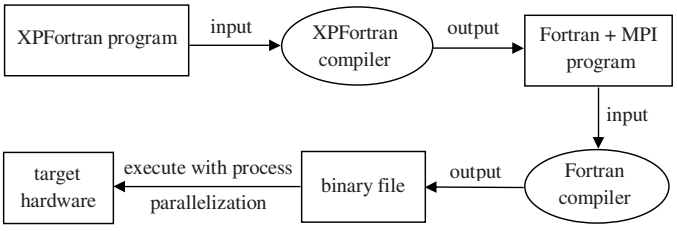


Fig. 4. Compilation and execution of XPFortran program

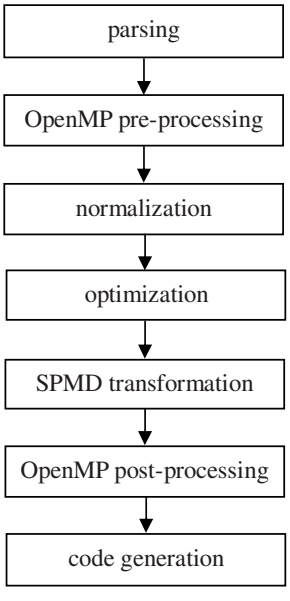


Fig. 5. Processing flow of XPFortran compiler with OpenMP support

The SPMD transformation phase converts the intermediate code which describes the behavior of the entire system into the code describing the behavior of each processor. For example, the range of loop indices and the bounds of array variables are converted from the global name-space to the local name-space. The code generation phase transforms the intermediate code into Fortran programs and MPI calls. Since there is no call to special communication library, the output of XPFortran compiler can be compiled by any Fortran compiler and run on any SMP cluster platform as long as standard Fortran and MPI are supported.

2.4 SPMD Implementation of XPFortran Compiler

Our XPFortran compiler implements the SPMD transformation of XPFortran program by compounding the following two transformations:

1. Index localization of variables.
   The following procedures are executed for a partitioned local variable in the partitioned axis:
   1.1 Narrow down the declared size of the variable to that after partitioning.
   1.2 Convert the subscripts in the reference of the variable to the local indices using function *gtol*.
   gtol is a function `k=`*gtol*`(I[,p])`, in which `I` and `k` are global and local indices respectively. `p` is the process number.
2. Index localization of computational loops.
   The following procedures are executed for the loops parallelized with `SPREAD DO` directives:
   2.1 Convert global loop indices to local loop indices using function *gtol2* or function *gtol3*. Loop indices can be pair of initial and final values, or triplet of initial value, final value and increment.
   gtol2 and *gtol3* are functions of loop parameters and mapping of loop to processors with rank of processor.
   2.2 Convert the loop variables in a loop body to the global indices using function *ltog*.
   ltog is a function `I=`*ltog*`(k,p)`, in which the meanings of `I`, `k` and `p` are same as those in *gtol*.

Table 1 shows the transformations executed by functions *gtol* and *ltog*. In the table, `w` is the chunk size. `block` means a variable or loop is divided equally and `cyclic` means it's divided in a round robin way with `w=1`. `P` is the number of processes. `N` is the size of the variable or loop before distribution.

Table 1. Global-local transformation and size after distribution

Type of Distribution	`k=`*gtol*`(I[,p])`	`I=`*ltog*`(k,p)`	Size after Distribution
`block`	`I mod w`	`k+p*w`	`w`
`cyclic`	$\lfloor \frac{I}{P} \rfloor$	`k*P+p`	$\lceil \frac{N}{P} \rceil$

Fig. 6 illustrates an example of the above transformations. Please note that the expression of `AL(I)` remains `AL(k)` after the transformations because of the compound transformation *gtol*(*ltog*(`k`)). Here *ltog*(`k,p`) is `(k+p*25)`, and *gtol*(*ltog*(`k,p`)) is `((k+p*25) mod 25)`. Since the local index `k` is between `1` and `25`, *gtol*(*ltog*(`k,p`)) becomes `k`.

The equations in Table 1 executed by functions *gtol* and *ltog* are integrated into the SPMD Fortran code generated by XPFortran compiler. In most cases it's also possible to express functions *gtol2* and *gtol3* in several or tens of Fortran statements, as shown in Fig. 6(b).

It's also important to note that code generated for this SPMD transformation is local to each processor and thread-safe. Therefore we draw the following conclusions:

– For the hybrid parallelization of nested loops executed by `SPREAD DO` directive and OpenMP `DO` directive, it's possible to use `DO` directive for outer loops and `SPREAD DO` for inner ones if there is no data dependence in the variables of the loops parallelized by the directives.
– For the SPMD transformation of nested `SPREAD DO` constructs, the generated code of *gtol2* and *gtol3* for all `SPREAD DO` constructs can be put just before the outmost loop if there is no data dependence between variables of loops the directives parallelize. Because of this, cost for the transformation of inner `SPREAD DO` constructs can be much reduced.

```
      INTEGER AL(100)
!XOCL PROCESSOR P(4)
!XOCL LOCAL AL(/(P))
!XOCL SPREAD DO
      DO I=1,100
         ... I ...
         ... AL(I) ...
      END DO
!XOCL END SPREAD
```

(a) XPFortran program

```
      ...
INTEGER AL(25)                    ! 1.1
spmdX0=ORG_RANK                   ! start of 2.1
spmdX1=25*spmdX0
IF (spmdX0.LE.0) THEN
  spmd_startX0=-spmdX1
ELSE
  spmd_startX0=0
ENDIF
IF (spmdX0.LT.3) THEN
  spmd_endX0=24
ELSE
  spmd_endX0=99-spmdX1
ENDIF
DO k=spmd_startX0,spmd_endX0      ! end of 2.1
   ... k+25*ORG_RANK ...          ! 2.2
   ... AL(k) ...                  ! 1.2&2.2
END DO
```

(b) Output SPMD program

Fig. 6. Example of SPMD transformation for `SPREAD DO` construct

3 Process-Thread Hybrid Programming with XPFortran and OpenMP

XPFortran supports OpenMP as a way of thread parallelization. OpenMP directives can be used inside some XPFortran constructs, forming "process outside, thread inside" hybrid programming style which is common sense to almost all programmers of parallel programs. Moreover, we have also considered using some XPFortran directives inside OpenMP constructs, which might beyond many people's expectation. Our motivation is as follows.

In XPFortran-OpenMP hybrid program, for nested loops which commonly exist in scientific applications, it's desired to use both XPFortran and OpenMP directives to parallelize the outmost loop. However, when the number of processors is huge, as in nowadays peta-scale and near-future exa-scale systems, hundreds of thousands or even millions of nodes exist, the grain of parallelization

is too fine to derive satisfying performance when all processes and threads are used to parallelize a single loop, the outmost one. Therefore it's necessary to parallelize both the outmost loop and inner ones in nested loops.

When one considers using process-thread hybrid parallelization for such nested loops, it's common sense to choose process parallelization for outer loops and threads for inner ones. However, because of problems such as the heavy overhead of run-time library calls to initialize thread environment and memory access discontinuity, it usually induces poor performance when thread parallelization is used for inner loops. The overhead to initialize MPI environment is usually also heavy. However, we have realized a light-weight implementation as introduced in section 2.4.

Because of the above reasons, it's necessary to think over how to deploy processes and threads for hybrid execution on large scale systems. We have considered using OpenMP `DO` or `PARALLEL DO` directives for outer loops of nested loops and XPFortran `SPREAD DO` for inner ones to improve performance. It's sure that they can't be combined with full flexibility. At least, thread spawn can not happen before the initialization of MPI environment. Also, for portability, we consider MPI environment with the lowest level of thread safety support, that is, `MPI_THREAD_SINGLE` in MPI standard. Because of this consideration, no MPI library routine can be called from OpenMP thread parallel section.

In the following sections, we introduce our language support and compiler implementation of OpenMP used in XPFortran program, and discuss some variants of XPFortran-OpenMP hybrid programming.

3.1 Language Support

In XPFortran, thread parallelization is supported by using OpenMP or automatic parallelization. For OpenMP, all OpenMP directives can be used in XPFortran constructs in which there is no communication. For example, OpenMP directives can be used in a `SPREAD DO` construct only if the construct doesn't contain any access to global variables, which causes communication between processors. Some XPFortran directives such as `SPREAD DO` can also be used in OpenMP constructs as long as no MPI communication happens. Because of this flexibility to describe the hybrid programs, there are some variants of XPFortran-OpenMP hybrid programming. For example, for the nested loops that can be parallelized, there are some choices to which loop process or thread parallelization is used for. It's also possible to use both process and thread parallelization for the same loop, as shown in Fig. 10(a).

Fig. 7 shows example of how to describe 2-D XPFortran and OpenMP hybrid programming in different ways for nested loops.

In Fig. 7(b) and 7(c), process parallelism realized by XPFortran `SPREAD DO` directives is executed within thread parallelism realized by OpenMP `PARALLEL DO` directive. There is no problem with this since there is no data dependence between the loop variables `I`, `J`, and `K`. Also, there is no MPI routine called from the OpenMP construct. Code generated for the three cases is shown in Fig. 8.

```
!XOCL PROCESSOR P(4,4)
!XOCL SPREAD DO /(P.1)
      DO K=1,KUB
!XOCL SPREAD DO /(P.2)
        DO J=1,JUB
!$OMP PARALLEL DO
          DO I=1,IUB
            A(I,J,K)=I
          END DO
        END DO
!XOCL END SPREAD
      END DO
!XOCL END SPREAD
```

(a)

```
!XOCL PROCESSOR P(4,4)
!XOCL SPREAD DO /(P.1)
      DO K=1,KUB
!$OMP PARALLEL DO
        DO J=1,JUB
!XOCL SPREAD DO /(P.2)
          DO I=1,IUB
            A(I,J,K)=I
          END DO
!XOCL END SPREAD
        END DO
      END DO
!XOCL END SPREAD
```

(b)

```
!XOCL PROCESSOR P(4,4)
!$OMP PARALLEL DO
      DO K=1,KUB
!XOCL SPREAD DO /(P.1)
        DO J=1,JUB
!XOCL SPREAD DO /(P.2)
          DO I=1,IUB
            A(I,J,K)=I
          END DO
!XOCL END SPREAD
        END DO
!XOCL END SPREAD
      END DO
```

(c)

Fig. 7. Some variants of 2-D XPFortran and OpenMP hybrid programming

```
      ...
  (K1,K2)=gtol2(1,KUB)
  DO K=K1,K2
   (J1,J2)=gtol2(1,JUB)
   DO J=J1,J2
!$OMP PARALLEL DO
    DO I=1,IUB
      A(I,J,K)=ltog(I)
    END DO
   END DO
  END DO
```

(a)

```
      ...
  (K1,K2)=gtol2(1,KUB)
  DO K=K1,K2
!$OMP PARALLEL DO
   DO J=1,JUB
    (I1,I2)=gtol2(1,IUB)
    DO I=I1,I2
      A(I,J,K)=ltog(I)
    END DO
   END DO
  END DO
```

(b)

```
      ...
!$OMP PARALLEL DO
  DO K=1,KUB
   (J1,J2)=gtol2(1,JUB)
   DO J=J1,J2
    (I1,I2)=gtol2(1,IUB)
    DO I=I1,I2
      A(I,J,K)=ltog(I)
    END DO
   END DO
  END DO
```

(c)

Fig. 8. Output of XPFortran compiler for Fig. 7

Note that all code for *gtol2* transformation in all the three cases can actually be put before loop `K` by using optimization introduced in the end of section 2.4.

3.2 Compiler Implementation

Compilation and execution process of XPFortran-OpenMP hybrid program is shown in Fig. 9. Like Fortran compiler with OpenMP support, our XPFortran compiler provides options `-Kopenmp` and `-Knoopenmp` to specify whether OpenMP directives in XPFortran program are effective or not. If `-Kopenmp` is specified, the directives will be compiled according to XPFortran and Fortran grammars regarding OpenMP. If `-Knoopenmp` is specified, the directives will be treated as comment lines. By default `-Knoopenmp` is effective.

Inside the XPFortran compiler, if `-Kopenmp` option is specified, the parser transforms the OpenMP directives to a data structure called `Comment_block`, which is intermediate representation for comment lines, as shown in Fig. 10(b). The parser only analyzes sentinels of OpenMP directives. By default comment

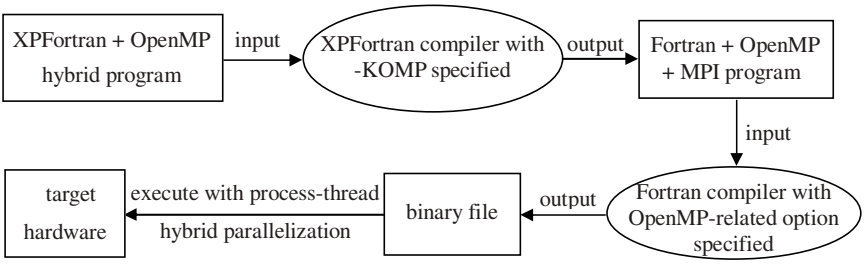


Fig. 9. Compilation and execution of XPFortran-OpenMP hybrid program

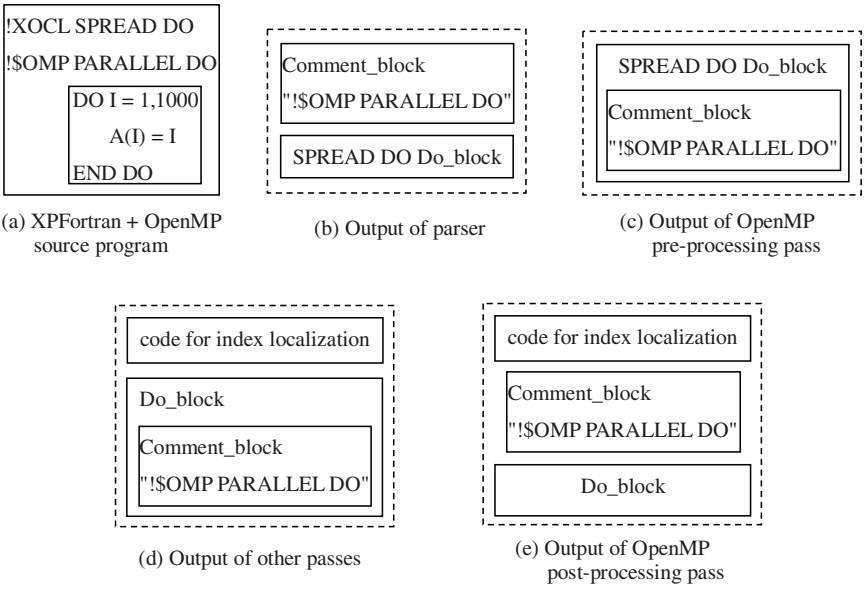


Fig. 10. An example of processing of OpenMP directives in XPFortran compiler

lines are not output by the XPFortran compiler, so if the sentinels are the ones of the OpenMP directives, the line is output by parser as `Comment_block`. Otherwise the line is deleted. The pass for OpenMP pre-processing analyzes what kind of OpenMP directive the line is, and then check errors according to XPFortran and OpenMP grammars. For example, in XPFortran no global variable can be used in OpenMP construct. Error message is output if global variable does appear in an OpenMP construct. Also, according to OpenMP specification, OpenMP `DO` or `PARALLEL DO` directive must be followed by do loop. The pass outputs error message if no do loop follows such directive. The pass also executes some operations to ensure the following passes don't exert side effects to OpenMP directives. For example, if `SPREAD DO` and `PARALLEL DO` directives are used for the same loop, as in Fig. 10(a), the pass for SPMD transformation

will output codes for loop localization immediately before the loop, as shown in Fig. 6(b) and Fig. 10(d). This will separate `PARALLEL DO` directive and the loop, which violates OpenMP grammar. To avoid this, the pass will save the line of OpenMP directive to the intermediate representation of the loop, the `Do_block`, as shown in Fig. 10(c). The pass for OpenMP post-processing will move the OpenMP line out of the Do_block, as shown in Fig. 10(e).

4 Performance Evaluation of XPFortran-OpenMP Hybrid Programming

This section describes our performance evaluation results of XPFortran-OpenMP hybrid programming. First we evaluate the overheads of some OpenMP and XPFortran directives, and then evaluate the hybrid parallelization on the application called Himeno benchmark program[2]. We use our XPFortran compiler to compile XPFortran program, and Fujitsu Fortran compiler 8.1 to compile OpenMP program and Fortran code generated by XPFortran compiler. Also we use Fujitsu MPI-2 7.2 library.

We run the evaluations on Fujitsu FX1. Each node is SPARC64 VII with 4 cores and the nodes are connected by high-speed Infiniband network. Each node has maximum 32GB memory and the memory bandwidth is 40Gbps.

4.1 Overheads of OpenMP and XPFortran Directives

We evaluate the overheads of OpenMP `PARALLEL`, `DO`, `PARALLEL DO` and XPFortran `SPREAD DO` directives since they are the most commonly used directives in hybrid execution. The overheads of the OpenMP directives are evaluated with EPCC OpenMP Microbenchmarks[3]. The overhead of XPFortran `SPREAD DO` is evaluated in a similar way. The number of threads and processes varies from 1, 2, to 4.

The results are shown in Table 2. Please note that in the table the result of `SPREAD DO` is that of the directive with `NOBARRIER` clause specified. That is, the cost of `SPREAD DO` is that when there is no barrier among processes. The barrier at the entrance or the end of a `SPREAD DO` construct is often not necessary since in XPFortran point-to-point communication other than one-sided communication is used in most cases.

As is shown, the overhead of `DO` is much lower than that of `PARALLEL DO`. This means that if thread parallelization is used for a nested loop, the `PARALLEL` directive should be put as outer as possible to reduce the cost of the initialization of thread environment. Also, from the evaluation we can see that with the same number of threads and processes, the overhead of `SPREAD DO` is lower than that of the OpenMP `PARALLEL` and `PARALLLEL DO` directives. Both the overhead of `SPREAD DO` and that of OpenMP `DO` with `NOWAIT` clause specified are low enough to be neglected.

Because of the low overhead of `SPREAD DO`, it's possible to use it to parallelize an inner loop in nested loops if either `SPREAD DO` or OpenMP `(PARALLEL) DO`

Table 2. Cost of some OpenMP and XPFortran directives

Directive	Number of threads or processes		
	1	2	4
OpenMP PARALLEL	0.18μs	0.51μs	0.76μs
OpenMP DO	0.02μs	0.21μs	0.39μs
OpenMP DO(NOWAIT clause specified)	0.02μs	0.02μs	0.04μs
OpenMP PARALLEL DO	0.19μs	0.56μs	0.83μs
XPFortran SPREAD DO (NOBARRIER clause specified)	0.007μs	0.007μs	0.009μs

must be used to parallelize the loop. Certainly the directive overhead is not the only factor that influences the performance of loops, therefore in the next section we evaluate the overall performance of different combinations of XPFortran-OpenMP hybrid parallelization.

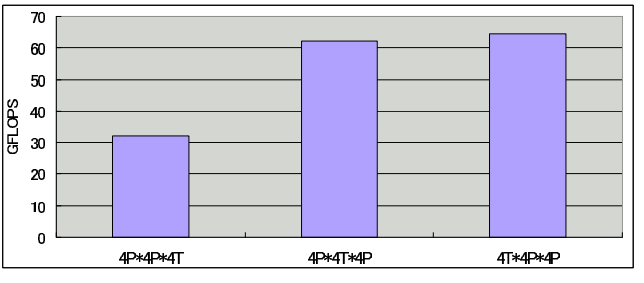


Fig. 11. Performance comparison when XPFortran and OpenMP directives parallelize different loops in the nested loops

4.2 Evaluation of Hybrid Execution on Himeno Benchmark Program

Himeno benchmark program is a Poisson equation solver using Jacobi iteration method. It measures the speed of main loops in the solver in MFLOPS. The characteristic of Himeno benchmark is having shift communication between neighbor nodes, which can often be found in data parallel programs, and 3-D nested loops that can be parallelized. The numbers of iterations in the outermost, middle and innermost loops of the 3-D nested loops are 256, 256, and 512 respectively.

We modified 1-D XPFortran version Himeno program downloaded from [2] and made 2-D XPFortran-OpenMP hybrid version.

As shown in Fig. 7, there is a variety of ways for XPFortran-OpenMP hybrid programming. In Fig. 11, we compared the performance of these methods. We used 2-D XPFortran-OpenMP hybrid programs. The number of processes is 16 and that of threads is 4 for all three cases. These cases correspond to programs in Fig. 7(a), (b), and (c) respectively, with the computation in the innermost loop

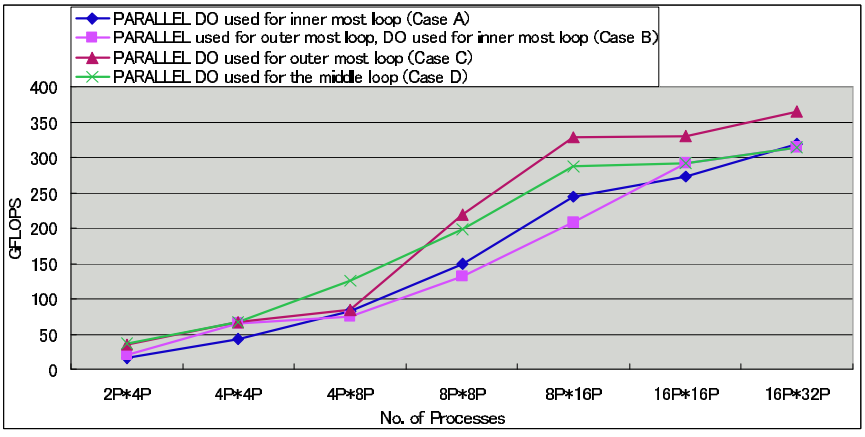


Fig. 12. Performance comparison when OpenMP PARALLEL and DO directives are used to parallelize different loops in the nested loops

being much more complex. For simplicity, we changed the unit of measurement from MFLOPS to GFLOPS. As shown in Fig. 11, for Himeno the performance is the best when the outermost loop is parallelized by OpenMP.

The difference in the overheads of `SPREAD DO` and `PARALLEL DO` is a reason for the difference in performance. To find the other reasons, we fixed the loops that `SPREAD DO` directives parallelize, and compared the performance when `PARALLEL` and `DO` directives are used to parallelize different loops of nested loops in 2-D XPFortran-OpenMP hybrid program, with varying number of processes. The result is shown in Fig. 12. For the nested loops, `SPREAD DO` directives are used to parallelize the outer two loops. For example, 2P*4P in the horizontal axis means 2 and 4 processors are used to parallelize the outermost loop and the middle loop respectively. With the number of processes fixed, loop(s) which `PARALLEL` and `DO` directives parallelize varies. Number of threads used is 4.

As the figure shows, for the cases A, C, and D, the performance of C is the best almost for any number of processes. This means that OpenMP `PARALLEL DO` directive should be used for the outermost loop as much as possible. Comparing case A with case B, we can see that their difference is not obvious. This is because that as the number of processes increases, the number of iterations executed by each process and parallelized by threads decreases, though there is a difference in the overheads of `DO` and `PARALLEL DO` directives just before the innermost loop, and the iterations in the outermost and middle loops aggravate such difference. This shows that the difference in the overheads is not the main reason of the difference in performance. Therefore performance depends mainly on which loop the `DO` directive parallelizes. According to profiling information for cases in Fig. 11, the data access cost ratio, which is the ratio of the data access time to the execution time of application, is 28.8%, 11.1% and 10.0% respectively for the three cases. The poor memory access efficiency in the first case worsens its performance greatly. This is because there are lots of accesses

to multi-dimensional arrays in the equation in Himeno program. When the outermost loop is thread parallelized, memory access is continuous, while it is not in the other two cases.

From the above evaluations, we can see that in some cases to use `SPREAD DO-PARALLEL DO` hybrid parallelization for nested loops, it's better to use `PARALLEL DO` outside and `SPREAD DO` inside to improve performance.

5 Conclusion

Because of the shared-distributed hybrid characteristic of memory in SMP clusters, process-thread hybrid programming is commonly used to parallelize applications running on such systems to derive high performance. XPFortran, a high-level programming language for process parallelism, is much easier to program than MPI because of its global address space and directive programming style. In this paper we introduced XPFortran-OpenMP hybrid programming.

As we have shown, XPFortran and OpenMP hybrid programming can be realized in various ways. To use thread parallelism for nested loops that can be parallelized, usually parallelization should be done for outer loop(s) to deploy data continuity. Also, as we have shown, the overhead of XPFortran data-parallel directive is lower than that of corresponding OpenMP directives, so in XPFortran we made it possible to describe process parallelization inside thread parallelization providing some conditions meet, namely to use some XPFortran directives inside OpenMP constructs.

Acknowledgements

The authors would like to acknowledge Japan Aerospace Exploration Agency (JAXA) for its contribution to performance evaluation in this paper. The authors also would like to thank Dr. Larry Meadows and the reviewers for their valuable advices and suggestions to improve the paper.

References

1. Co-Array Fortran, `http://www.co-array.org/`
2. Himeno Benchmark Program, `http://accc.riken.jp/HPC_e/HimenoBMT_e.html`
3. EPCC OpenMP Microbenchmarks, `http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html`
4. Message Passing Interface Forum. MPI: A message-passing interface standard, `http://www.mpi-forum.org/`
5. OpenMP, `http://openmp.org/wp/`
6. Iwashita, H., Aoki, M.: Mapping Normalization Technique on the HPF Compiler fhpf. In: Labarta, J., Joe, K., Sato, T. (eds.) ISHPC 2005 and ALPS 2006. LNCS, vol. 4759, pp. 315–329. Springer, Heidelberg (2008)
7. Iwashita, H., Sueyasu, N., Kamiya, S., van Waveren, M.: VPP Fortran and the Design of HPF/JA Extensions. In: Concurrency and Computation: Practice and Experience, vol. 14(8-9), pp. 575–588. John Wiley &Sons Ltd., Chichester (2002)

8. Iwashita, H., Aoki, M.: A Code Generation Technique Common to Distribution Kinds on the HPF Translator. IPSJ Transactions on Advanced Computing Systems (ACS) 0(15), 329–339 (2006) (in Japanese)
9. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proc. of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
10. Benkner, S., Sipkova, V.: Language and Compiler Support for Hybrid-Parallel Programming on SMP Clusters. In: Zima, H.P., Joe, K., Sato, M., Seo, Y., Shimasaki, M. (eds.) ISHPC 2002. LNCS, vol. 2327, pp. 11–24. Springer, Heidelberg (2002)

A Case for Including Transactions in OpenMP

Michael Wong[1], Barna L. Bihari[2], Bronis R. de Supinski[2],
Peng Wu[1], Maged Michael[1], Yan Liu[1], and Wang Chen[1]

[1] IBM Corporation
[2] Lawrence Livermore National Laboratory
{michaelw,yanliu,wdchen}@ca.ibm.com,
{bihari1,bronis}@llnl.gov,
{pengwu,magedm}@us.ibm.com

Abstract. Transactional Memory (TM) has received significant attention recently as a mechanism to reduce the complexity of shared memory programming. We explore the potential of TM to improve OpenMP applications. We combine a software TM (STM) system to support transactions with an OpenMP implementation to start thread teams and provide task and loop-level parallelization. We apply this system to two application scenarios that reflect realistic TM use cases. Our results with this system demonstrate that even with the relatively high overheads of STM, transactions can outperform OpenMP critical sections by 10%. Overall, our study demonstrates that extending OpenMP to include transactions would ease programming effort while allowing improved performance.

1 Introduction

Many have observed that Transactional Memory (TM) could simplify shared memory programming substantially by simply marking a group of load and store instructions to execute atomically, rather than using locks or other synchronization techniques. TM's promise of easier program understanding, along with composability and liveness guarantees has led to a fad status for TM. As a result, extensions to OpenMP [6] to include transactions have received interest in the OpenMP community [5,8]. However, we must first determine whether TM can be more than just a research toy before these extensions can receive serious consideration.

Two implementation strategies are available for TM. Hardware TM (HTM) modifies the memory system, typically through modifications to the L1 cache, to support atomic execution of groups of memory instructions. Software Transactional Memory (TM) provides similar functionality without using special hardware. In this paper, we combine an STM system that provides the transaction primitive with an OpenMP implementation that provides all other shared memory functionality. The combination is natural since the TM system relies on compiler directives that are similar to OpenMP's syntax.

The remainder of this paper is organized as follows. We present our STM system in Section 2. We then review its integration with a production-quality OpenMP compiler in Section 3. In Section 4, we present performance results for two application scenarios, which demonstrate that transactions can improve performance by 10% over a production quality critical section implementation even with the relatively high overhead of

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 149–160, 2010.

STM. Overall, we conclude that extending OpenMP to include transactions would reduce programming effort while supporting potential performance gains.

2 The IBM XL STM Compiler

Generally, an STM system uses a runtime system to manage all transactional states. This runtime annotates reads and writes for version control and conflict detection. If two transactions conflict, (i. e., the write set of one intersects with the read set of the other), the system may delay or abort and retry one of them. The system validates the reads at the end of the transaction (i. e., any loaded values have not changed). The system then commits the writes (i. e., stores them to their actual memory locations) if it does not detect any conflicts. Compared to HTM, STM does not require special hardware while enabling scalability of inherently concurrent workloads at the cost of higher overhead, particularly when no conflicts occur.

IBM released a freely downloadable STM compiler under alphaWorks® site [7] based on its production IBM® XL C/C++ Enterprise Edition for AIX®, Version 9.0, called IBM XL C/C++ for Transactional Memory for AIX in 2008. This implementation includes standard and debugging versions of the runtime libraries. Fig. 1 shows the TM features [4] of our XL STM compiler.

STM characteristics	IBM STM compiler
Isolation	Weak
Granularity	8 byte block
Direct/Deferred Update	Deferred or Lazy
Concurrency Control	Optimistic
Synchronization	Blocking
Conflict Detection	Early(read after write) Late (write after write)
Inconsistent Reads	Tolerated(signals and infinite loop checks)
Conflict Resolution	Polite
Nested Transaction	Flat
Exception	terminate

Fig. 1. Transactional features of XL STM

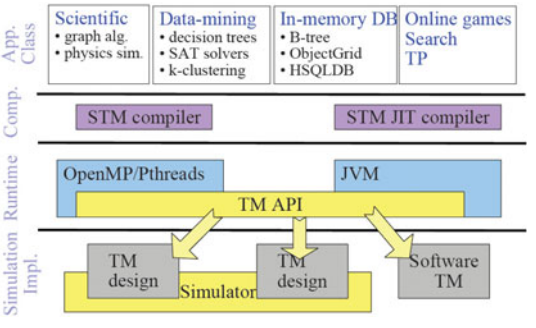


Fig. 2. Project Components of our STM that rely on OpenMP

Our STM uses a block-based mapping of shared memory locations, which enables support for different languages, unlike the alternative object-based mapping. The system buffers writes, which are written to the global address space only when the transaction is guaranteed to commit. When a transaction writes to a stack location or privately allocated memory that has not yet escaped the thread (i.e., a memory location that is not yet visible to other threads), the write does not induce any conflict. These contention-free writes do not require write-barriers, but may need to be memory checkpointed if the system must recover the overwritten values upon a retry [9].

Using data-flow analysis, the compiler can exclude writes to most contention-free locations from memory checkpointing. Basically, the write requires no checkpointing if

the variable or heap location is private to a transactional lexical scope (e.g., transactional block or procedure), that is, the variable is not live upon entry and exit to the lexical scope. However, contention-free writes often have uses after exiting the transaction. In this case, we can still avoid checkpointing if the location is not live upon entry to the transactional scope the write to the location dominates the transaction end. The latter property guarantees that the location will always be re-defined upon retry and, thus, we do not need to recover the original value. Finally, uninitialized locations upon a transaction entry require no checkpointing. This also includes the case when a heap location is allocated within the transaction.

The read barrier does not write to shared meta-data. The checkpoint barrier records the original value for writes to contention-free locations. The compiler implements retries with `setjmp` and `longjmp`. In addition, our STM system focuses on instrumentation statistics to identify STM bottlenecks in applications, thus avoiding time consuming and error-prone manual instrumentation.

We use C to implement the runtime system, the source code of which is freely available as part of an Open Source Amino project [2]. This runtime supports exploration of a range of TM scenarios (HTM, STM and hardware accelerated STM) in multiple languages including C, C++ and Java^{TM} as Fig. 2 shows. The current implementation assumes *weak isolation*: accesses to a particular shared location always occur within transactions (a *transactional location*) or never within them. Further, it assumes that transactions only include any revocable operations without side effects (e. g., no file I/O) and, hence, we can safely undo and retry them.

Our runtime uses metadata to synchronize transactional access to shared memory locations. We associate a metadata entry with transactional location. This entry includes a version number for tracking updates of the location and a lock to protect updates of it. A thread can write to memory only if it holds the associated metadata lock. A transaction increments the version number when it releases the metadata lock, which guarantees that the data has not changed if the version number is unchanged. Thus, a transaction can read a metadata version number and then the associated data and then later check the version number to determine if the data is unchanged.

Our runtime maintains information for each thread that includes its read set, write set, statistics, status, level of nesting, and lists of mallocs, frees and modified local variables. The read set information is the metadata location and version number. The write set information contains the address, value, size and metadata location.

When a thread begins a transaction, it sets its transactional status data. It also reads some global data in some configurations. The thread then records the current version number before it reads a transactional location. In some configurations, the thread also checks the consistency of the read set. For transactional writes, the thread records the target address and the value to be written. In some configurations, it acquires the corresponding metadata lock; otherwise, it acquires metadata locks for all locations in its write set at the end of the transaction. When the transaction ends, the thread then validates the consistency of its read set. If that fails, the thread aborts the transaction by releasing the metadata locks and jumping to the beginning of the transaction. Otherwise, the thread writes the values to the addresses in its write set, releases the metadata locks and then resets its transactional status data.

Table 1. STM runtime statistics

Statistic	Description
READ_ONLY_COMMITS	Number of committed transactions with no writes
READ_WRITE_COMMITS	Number of committed transactions with writes
TOTAL_COMMITS	Number of successfully committed transactions
TOTAL_RETRIES	Number of retried transactions
AVG_RETRIES_PER_TXN	Average number of retries per committed transaction
MAX_NESTING	Maximum level of transaction nesting
READ_SET_SIZES	Unique locations in read sets of committed transactions
WRITE_SET_SIZES	Unique locations in write sets of committed transactions
AVG_READ_SET_SIZE	Average number of unique locations in read set per transaction
AVG_WRITE_SET_SIZE:	Average number of unique locations in write set per transaction
READ_SET_MAX_SIZE	Maximum number of unique locations in a read set
WRITE_SET_MAX_SIZE	Maximum number of unique locations in a write set
READ_LIST_MAX_SIZE	Maximum number of locations in a read list
WRITE_LIST_MAX_SIZE	Maximum number of locations in a write list
DUPLICATE_READS	Number of transactional reads of locations previously read in the same transaction
PCT_DUPLICATE_READS	Percentage of transactional reads of locations previously read in the same transaction
DUPLICATE_WRITES	Number of transactional writes to locations previously written in the same transaction
PCT_DUPLICATE_WRITES	Percentage of transactional writes to locations previously written in the same transaction
NUM_SILENT_WRITES	Number of transactional writes of already stored value
PCT_SILENT_WRITES	Percentage of transactional writes of already stored value
READ_AFTER_WRITE_MATCHES	Number of transactional reads that follow a transactional write of the same location in the same transaction
PCT_READ_AFTER_WRITE	Percentage of transactional reads that follow a transactional write to the same location in the same transaction
NUM_MALLOCS	Number of calls to `malloc` inside transactions
NUM_FREES	Number of calls to `free` inside transactions
NUM_FREE_PRIVATE	Number of calls to `free` blocks allocated in that transaction

The STM runtime can collect statistics related to a program's inherent transactional characteristics (i. e., independent of our STM implementation), such as transaction sizes. It can also track implementation specific data, such as metadata locks acquired. We collate these statistics by static transactions (i. e., source code file name and first line number). We also generate aggregate statistics. Collecting these statistics incurs a significant performance cost but can guide optimization of TM programs.

The program must call `stm_stats_out` in order to generate the statistics files. We allow multiple calls to `stm_stats_out`. The statistics can be inconsistent if the call occurs while any transactions are active since the statistics can change during the snapshot. We recommend only calling `stm_stats_out` when only the main thread is active, which ensures the statistics are consistent. The `stats.h` file has a full list of the statistics; we provide some of the most important ones in Table 1.

We have ported our STM runtime to several platforms including AIX and Linux® on IBM PowerPC®, LinuxX86, and LinuxX86_64, in 32 or 64bit mode. IBM XL C/C++

for Transactional Memory for AIX generates code for this interface for programs that use the high-level interface that we discuss in the next section. Other compilers could also use our open source STM runtime for this interface.

3 IBM STM Compiler Design

3.1 Syntax

Programs define transactions for our STM compiler [3] through a simple directive:

```
1 #pragma tm atomic [default(trans|notrans)]
2   {
3
4   }
```

in which `default(trans|notrans)` defines the default behavior of memory referenced in the lexical scope of the transactional region. If the user specifies `default(trans)` then the compiler translates references to shared variables in the transactional region to STM runtime calls, often referred to as STM read-write-barriers, that ensure the correctness of the execution. If the user specifies `default(notrans)` then the compiler does not translate any memory references in the region to STM read-write-barriers. The `default` clause is optional; the default value is `trans`.

The code region encapsulated within our transactional construct is basically a structured block although the block cannot include any OpenMP constructs. We impose this restriction since otherwise an aborted transaction could lead to an inconsistent state for the OpenMP runtime library, such as acquiring a lock that it will never release.

Fig. 3 shows an example transactional region within an OpenMP parallel region. The compiler inserts `stm_begin()` and `stm_end()` around the transactional region. These calls allow our STM runtime to monitor all shared memory access within the transactional region in order to ensure the transaction executes correctly.

3.2 Special Transactional Function Attributes

Users must annotate functions that are called within transactional regions so that the compiler can correctly transform memory references within the function to STM runtime barrier calls. We provide two function attributes for this purpose: `tm_func` and `tm_func_notrans`. The user specifies `__attribute__((tm_func))` with a function declaration to indicate that a transactional region can call the function so the compiler must transform its memory references to STM runtime calls. The user specifies `__attribute__((tm_func_notrans))` with a function declaration to indicate that transactional region can call the function but the compiler should not transform its memory accesses to STM runtime calls. Neither attribute is required if transactional regions cannot call the function. Fig. 4 shows an annotated function declaration.

Our transactional function attributes reflect two key design factors. First, they allow function calls within transactions without requiring the user to make major code modifications. Second, the attributes allow the compiler to check that the function call

```
1  int b[25], j;
2  int index[5] = {4,5,675,22,34};
3
4  for( j = 0 ;j<25; j++ )
5     b[j] = 0;
6
7  #pragma omp parallel for
8     {
9     for(j=0; j<25; j++)
10      {
11      ...
12
13      #pragma tm atomic
14         {
15         b[index[j]] = ...;
16         }
17      }
18    }
19 }
```

Fig. 3. Sample code of using #pragma tm atomic

```
1 int foo (int sum) __attribute__((tm_func));
2 int foo (int sum)
3 {
4 return ++sum;
5 }
```

Fig. 4. Sample code of annotating function attribute to a function

conforms to our restriction that transactional regions cannot include any OpenMP constructs. The compiler issues a warning for any unannotated functions that are called within a transactional region. The programmer must ensure that the call is safe with the default behavior that does not transform memory references to STM runtime calls. These attributes are only needed for STM; HTM implementations do not require them.

4 Experimental Results

We provide results that evaluate the potential for TM to benefit scientific computing. In particular, we consider the opportunities to use transactions in unstructured-mesh multi-physics simulation applications, which are widely used because of their geometric and architectural flexibility. These applications typically have many compute-intensive loops with complicated memory referencing patterns. Although these applications usually exhibit good scalability with MPI-based domain-decomposition, the trend toward systems built with multicore nodes motivates explorations of moving them to a hybrid OpenMP/MPI programming model. However, many users view the complexity of shared memory programming in general, and of ensuring data race freedom in particular, as a significant barrier. Transactions directly address this concern.

While no production HTM implementations currently exist, STM provides a mechanism to experiment with using transactions within these applications. Thus, we have implemented a Benchmark for Unstructured-mesh Software Transactional Memory

```
1 #pragma tm atomic default(trans)
2 {
3   gradient[cell_no_1] += incr;
4   gradient[cell_no_2] -= incr;
5 }
```

Fig. 5. Gradients accumulated within transaction in `compute_cell`

(BUSTM). BUSTM provides a simple code in which to explore the programming benefits of transactions and any performance implications as it mimics the algorithms and behavior of real unstructured mesh applications.

When we construct a benchmark of an application scenario, we must ensure that it captures the salient features of the target application space. Since we primarily focus on race conditions and the potential benefits of TM, we artificially generate memory conflicts in either a deterministic or random yet still controllable manner. That is, even with random conflicts, we can configure their probabilities indirectly through input parameters. The benchmark also must include rigorous error checking for example problems with known or independently computable answers to ensure that the threaded experiments (with or without transactions) execute correctly.

BUSTM meets these requirements and uses realistic unstructured mesh connectivity. It mimics the complex and flexible bookkeeping common to unstructured mesh applications. BUSTM can handle unstructured cells with an arbitrary number of faces. We have already used a wide range of cell types, including triangular prisms, hexahedra, tetrahedra and pyramids. Like real unstructures mesh applications, BUSTM cross references the basic unstructured mesh building blocks of **nodes**, **faces** and **cells** in almost all combinations, so that *indirect indexing* pervades the code. This indirect addressing leads to extensive synchronization requirements to use shared memory programming, which is exactly when TM should provide benefits. The remainder of this section explores the benefits of our STM implementation with both deterministic and probabilistic conflicts.

4.1 Deterministic Conflicts

Domain decomposition based on MPI message passing has provided successful parallelization of conservative finite volume schemes on unstructured grids. Our careful evaluation of their typical memory access patterns, however, found that conflicts may occur as cells are updated during the traversal of face-based loops. Although these conflicts rarely occur in practice, we cannot assume they do not occur. Since ignoring them would lead to incorrect results, we must protect them with some synchronization method. However, locks have relatively high overhead when conflicts are unlikely. Thus, TM may provide substantial performance benefits for this scenario.

Short of a full-blown CFD solver, we simulate face-by-face flux computations with the *numerical divergence* of a mesh-function that we define on an unstructured mesh in a cell-centered sense. This emulation computes the gradient of a function. If the function is constant, its gradient and, thus, divergence is zero. Fig. 5 shows the transaction.

```
#pragma omp parallel for
    for (i=0; i < max_face; i++){
      left_neighbor = left_cells[i];
      right_neighbor = right_cells[i];
      compute_cell(incr, left_neighbor);  //face increments left cell
      compute_cell(incr, right_neighbor); //face increments right cell
    }
```

Fig. 6. Threaded face loop that calls `compute_cell`

Since real finite-volume codes have significant computation per face, BUSTM loops over the faces, as Fig. 6 shows. Memory conflicts can occur if different faces update the same cell. They only actually occur, resulting in incorrect execution, if `compute_cell` does not use transactions or other synchronization such as a `critical` section, and two updates happen at the same physical time. Thus, the probability of conflicts is extremely low and many of our experimental runs had no conflicts.

Our experiements use a mesh of 119893 triangular prism cells arranged as a 2-D layer of 3-D cells. This mesh has 420060 faces (some quadrilateral, others triangular) and 123132 nodes. The total number of potential conflicts is fixed with this mesh although the actual number of conflicts varies between runs. We observe the number of resolved conflicts from the statistics available with our STM implementations.

Fig. 7 shows the number of conflicts resolved in each of 1000 runs is always less than ten and frequently zero. Fig. 8 shows the sum of the number of conflicts that our STM implementation resolves over 1000 runs versus the total number of errors committed (without STM) over 1000 runs for a range of thread counts. We observe many fewer STM retries than errors with the unsynchronized code. However, both exhibit similar trends with increasing thread counts and rarely occur relative to the number of potential conflicts. Despite the relatively small mesh (which *increases* the conflict probability), only 0.00042% of all cell updates incurred conflicts on 16 threads. No conflicts occur with just two threads (or one, which is clearly expected), which makes debugging the unsynchronized code more difficult. The number of conflicts increases significantly with the number of threads, with conflicts being fairly likely with 16 threads.

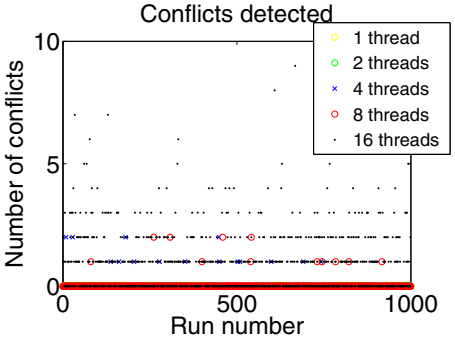


Fig. 7. Resovled deterministic conflicts

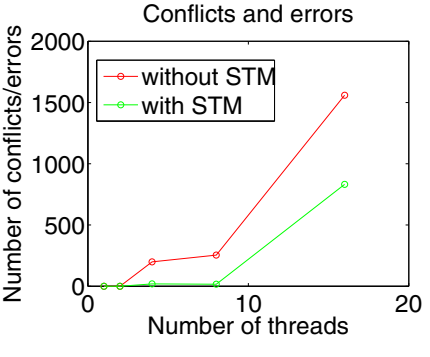


Fig. 8. Total conflicts or errors

```
1 #pragma tm atomic default(trans)
2 {
3   cell_counter[cell_no] ++;
4 }
```

Fig. 9. Cell counter incremented within transaction of `mark_cell`

4.2 Probabilistic Conflicts

Monte Carlo applications are another common type of scientific application in which memory conflicts potentially occur with threaded implementations. In these applications, randomly released particles travel through a computational mesh and increment a cell-based physical quantity each time they touch a cell. Parallelizing over the particles results in almost embarassingly parallel loops, except for the race condition produced by two particles (belonging to two different threads) trying to update the same cell at the same time. Although these memory conflicts are unlikely, some will occur with enough particles and threads.

We exploit the unstructured bookkeeping in BUSTM in order to emulate the behavior of particles without implementing a real Monte Carlo application. Instead of particles that travel along a straight line through space, they travel from cell to cell via the neighbor information available for each adjacent cell. Thus, our benchmark has two levels of randomness. First, we randomly select the cell in which the particle is "born". Second, we randomly choose the face that the particle exits the current cell.

We make sure the number of particles is independent of the number of cells. After being created, or "born," each particle is "alive" as long as it stays within the computational domain (i. e., the face through which it exits the current cell is an interior face). If that randomly selected face is a boundary face, it exits the domain and completes its path. This scheme results in a wide variance in the particle path lengths; some will have a short lifespan while others stay active within the domain for a long time. This property, which is consistent with real Monte Carlo simulations, limits scalability.

Fig. 9 shows the simple transaction that safely increments a single cell-based integer. Fig. 10 shows the loop that distributes the particles across the threads. As discussed above, each randomly generated particle moves through the mesh until it exits the domain. We also increment a separate counter for each particle so that we can compute the total number of touches without concern for potential conflicts as `cell_counter*particle_counter`. This check usually fails if we do not use any synchronization for the `mark_cell` calls.

Our probabilistic experiments use the same triangular prism mesh as the deterministics ones and we report conflict statistics observed by our STM implementation. We performed 100 runs, using 12000 random particles (10% of the number of cells) during each run. Fig. 11 shows a much higher number of conflicts than in the deterministic case. The conflicts are fairly consistent for a given thread count and appear almost linearly proportional to that count. We observe the opposite trend from the deterministic case between the total number of STM resolved conflicts and the total number of unsynchronized errors for this probabilistic test, as Fig. 12 shows. We find that STM incurs many more conflicts, sometimes by an order of magnitude. While we are still investi-

```
#pragma omp parallel for
for(i=0; i<max_particles; i++){
  next_cell = rand();
  while(inside){
    mark_cell(next_cell);   //particle increments cells
    next_face = rand();
    next_cell = neighbor(next_face);
    if(next_cell < 0)inside = 0;
  }
}
```

Fig. 10. Threaded particle loop that calls `mark_cell`

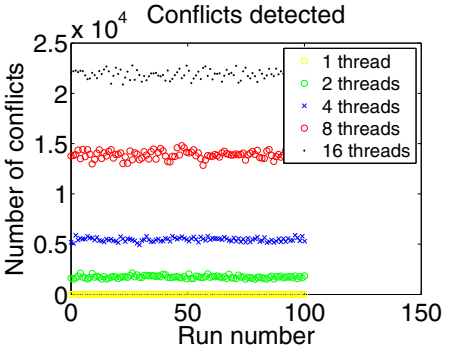


Fig. 11. Resolved probabilistic conflicts

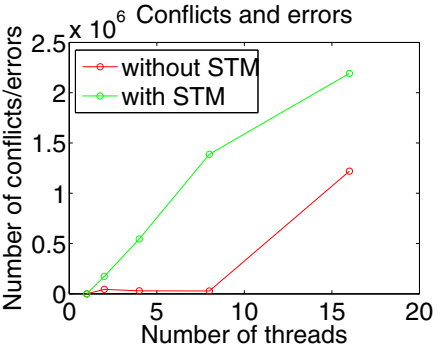


Fig. 12. Total conflicts or errors

gating this discrepancy, the much heavier computational load imposed by the frequent invocation of a random number generator may cause it. Nonetheless, the total number of resolved conflicts and committed errors is still small compared to the number of updates. For example, the conflict probability is only 0.009% on this relatively small mesh with 16 threads.

While the deterministic and probabilistic algorithms represent very different numerical algorithms and in their relationships of resolved conflicts to errors, they also have similarities. In both cases we have low conflict probabilities (much less than 0.01%) and both use the same unstructured mesh. Indeed, the results confirm our conjecture that the algorithms are well suited to transactions since conflicts rarely occur. We can observe reasonably good performance even with STM and expect higher performance with HTM since conflict resolution is generally the dominant cost with TM. Lock based implementations, on the other hand, would suffer much more overhead. Preliminary timing results that compare our STM implementations to ones that use the OpenMP `critical` construct indicate that STM provides about a 10% performance advantage despite the relatively large overheads of STM [1].

5 Conclusions and Future Work

We have presented the design and implementation of a software transactional memory system. Our system combines an open source runtime with modifications to the pro-

duction IBM® XL C/C++ Enterprise Edition for AIX®, Version 9.0 compiler. This system uses OpenMP to generate threads and parallelize applications. Transactions denoted by a simple directive serve as an alternative to OpenMP synchronization. Users also must annotate declarations of any functions accessed within a transaction. These annotations would be unnecessary with HTM support since they direct the compiler to transform memory accesses to STM primitives.

We evaluate the efficacy of TM for scientific computing. In particular, we develop a new TM benchmark, BUSTM, that explores the use of transactions for unstructured mesh applications. We present two distinct scenarios that BUSTM can emulate: CFD applications and Monte Carlo applications. In both cases, we find that conflicts for the transactions are infrequent; however, correct execution requires synchronization of some sort. Our initial performance results found that the STM implementation outperformed the equivalent OpenMP implementation with `critical` regions by 10%, a significant result considering that STM has relatively high overhead. Although we are continuing to explore these performance results with different meshes, we expect that the emulated scientific applications will benefit substantially from HTM support.

Acknowledgements

The authors wish to thank John Gyllenhaal and Scott Futral of LLNL for numerous fruitful discussions on this subject and for support of this work. The second author also acknowledges past financial support from Rockwell International, Boeing, and Hypercomp, Inc. in developing the unstructured mesh bookkeeping used in the experiments, and from Icon Consulting and IBM in writing the BUSTM code.

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DEAC52-07NA27344 (LLNL-CONF-422888).

References

1. Bihari, B.L.: Experiments Using IBM’s Software Transactional Memory Compiler (May 2009), `http://spscicomp.org/ScicomP15/slides/user/bihari.pdf`
2. IBM. Concurrent Building Block (May 2008), `http://sourceforge.net/projects/amino-cbbs/`

3. IBM. IBM XL C/C++ for Transactional Memory for AIX, V0.9 Language Extensions and Users Guide (May 2008), http://dl.alphaworks.ibm.com/technologies/xlcstm/xlcstm-whitepaper.pdf
4. Larus, J.R., Rajwar, R.: Transactional Memory (Synthesis Lectures on Computer Architecture). Morgan & Claypool Publishers, San Francisco (January 2007)
5. Milovanović, M., Ferrer, R., Unsal, O., Cristal, A., Martorell, X., Ayguadé, E., Labarta, J., Valero, M.: Transactional memory and openMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 37–53. Springer, Heidelberg (2008)
6. OpenMP ARB. OpenMP Application Program Interface, v. 3.0 (May 2008)
7. Wong, M.: IBM XL C/C++ for Transactional Memory for AIX (August 2009), http://www-949.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore/2009/08/11/ibms-alphaworks-software-transactional-memory-compiler
8. Woongki, B., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The OpenTM Transactional Application Programming Interface. In: PACT'07: Proceedings of the 16th International Conference on Parallel Architec-ture and Compilation Techniques, Washington, DC, USA, June 2007, pp. 376–587. IEEE Computer Society, Los Alamitos (2007)
9. Wu, P., Michael, M., von Praun, C., Nakaike, T., Bordawekar, R., Cain, H., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M., Wang, H.Y., Wang, K.: Compiler and Runtime Techniques for Software Transactional Memory Optimization. Journal of Concurrency and Computation: Practice and Experience, 7–23 (July 2009)

OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler

Satoshi Ohshima[1], Shoichi Hirasawa[2], and Hiroki Honda[2]

[1] The University of Tokyo/JST CREST, 2-11-16, Yayoi, Bunkyo-ku, Tokyo, Japan
ohshima@cc.u-tokyo.ac.jp
[2] The University of Electro-Communications/JST CREST, 1-5-1, Chofugaoka, Chofu-shi, Tokyo, Japan
{hirasawa,honda}@is.uec.ac.jp

Abstract. Arithmetic performance with GPGPU attracts attention. However, the difficulty of the programming poses a problem. We have proposed GPGPU programming which used the existing parallel programming technique. We are now developing OpenMP framework for GPU as a concrete of our proposal. The framework is based on Omni OpenMP Compiler and named "OMPCUDA". In this paper we describe a design and an implementation of OMPCUDA. We evaluated using test programs, and validated that parallel improvement in the speed could be easily carried out in the same code as the existing OpenMP.

1 Introduction

OpenMP attracts attention because of the tool of an inner node parallel programing environment of large scale SMP cluster as well as a programing environment for the shared memory parallel computers and multicore CPUs. On the other hand, GPU (Graphics Processing Unit) attracts attention as a new highly parallel high performance computational hardware. But the programming of GPU is difficult and complex because new parallel programing environments such as CUDA (CUDA Unified Device Architecture)[1] is necessary to make parallel programs running on GPU.

We aim at that more application programmers can use GPGPU simpler. Since GPU is suitable for parallel processing, applications with high parallelism is expected to improve the performance with GPGPU (General-Purpose computing on GPUs). We has proposed reduction of the time and effort for the application programmers by enabling use of the existing parallelization programming techniques at GPGPU programming [2].

As a concrete implementation of our aim, we are now developing an OpenMP framework for CUDA. The framework is based on Omni OpenMP Compiler[3] and named "OMPCUDA". This paper describes the design, implementation, and evaluation of OMPCUDA.

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 161–173, 2010.

2 Omni OpenMP Compiler and CUDA

2.1 Omni OpenMP Compiler

Omni OpenMP Compiler (henceforth OMNI) is an OpenMP compiler which developed in Tsukuba, JAPAN. OMNI does not support latest OpenMP specifications, but it has some useful features. Fig. 1 illustrates the overview of OMNI. OMNI is a translator which takes OpenMP programs as input, and generates the multithread C program with runtime library calls. The generated program is compiled by the native back-end compiler such as gcc and output executable file linked with the OMNI specific runtime library.

OMNI has a unique and useful intermediate code and a toolkit written in JAVA for handling the intermediate code. The intermediate code is called Xobject code. Also, OMNI has front-ends which convert from a program written in normal programming language to the Xobject code. So if there are corresponding front-ends, OMNI can handle various languages in Xobject code level using the toolkit. OMNI version 1.6a (stable version) now has C/C++ (C++ is incomplete) and Fortran77 front-ends.

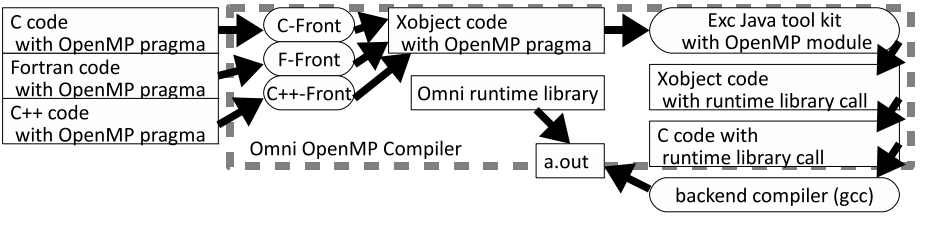


Fig. 1. Overview of Omni OpenMP Compiler

The main rolls of runtime library of OMNI are as follows:

1. Thread management: Thread management such as creating threads and destroying threads are the runtime library's roll. Moreover, procedures in threads such as reductions and barrier synchronizations are the runtime library's rolls too.
2. Loop rewriting: In the OMNI's simple loop parallelization, each thread executes a same function. But loop parameters – begin, end, and step – are rewritten by runtime library, so each threads can execute its own part of loop.

2.2 CUDA

CUDA is a framework for utilizing NVIDIA GPU as a data parallel computer developed by NVIDIA. It offers the GPGPU program development environment by the extended programming language of C/C++ and a specific compiler named

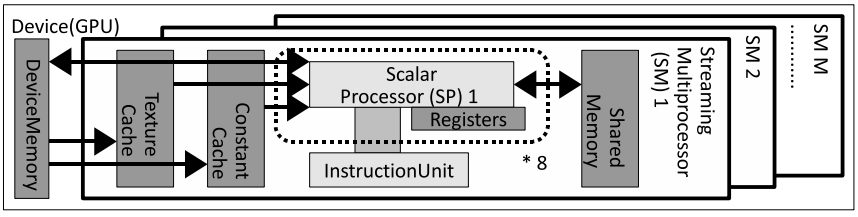


Fig. 2. Hardware model of CUDA-capable GPU

"nvcc". There is not so much difference between C/C++ and CUDA in language specification, but CUDA has unique specifications in a hardware model, a memory model, and an execution model.

The hardware model of CUDA-capable GPU is shown in Fig. 2. GPU contains operation units (computing-element group) called Streaming Multiprocessor (henceforth SM). The number of SMs is 1 to 30 in the current GPUs. SM has eight computing elements called Scalar Processor (henceforth SP). And there are Grid, Block, and Thread as a logical unit of parallel execution.

Grid is a unit at the time of CPU making GPU performs some operation. Only 1 Grid can be assigned to GPU at the same time. Grid consists of one or more Blocks (henceforth CUDA Block(s)). CUDA Block consists of one or more Threads (henceforth CUDA Thread(s)). The number of CUDA Blocks and the number of CUDA Threads per CUDA Block can be specified when Grid is executed on GPU. On this occasion, *number of CUDA Blocks* $\times$ *number of CUDA Threads* tasks are generated and they are assigned to SMs of CUDA Blocks.

Parallel execution of CUDA Threads in each CUDA Block is done by SPs in the same SM. The unique ID is assigned to each CUDA Block and each CUDA Thread at the beginning of the Grid execution. When the number of CUDA Blocks and CUDA Threads are more than the number of physical SMs and SPs, they are scheduled in the time-sharing manner. Because a scheduler of CUDA can switch the active CUDA Blocks and CUDA Threads in very low costs, the number of CUDA Blocks and CUDA Threads should be more than the number of physical processors to hide the latency of memory access. Parallel processing with high parallelism is a very important point of CUDA.

CUDA has several kinds of memory on GPU (Fig. 3), and each memory have specific features. In order to perform memory allocation and memory transfer between a main memory and a GPU memory, programmers must use CUDA specific API functions. In order to totally utilize GPU and get high performance, it is necessary to utilize various GPU memory types. It is a burden for application programmers.

In order to create high-performance programs using CUDA, programmers have to understand the architecture of GPU, the execution model, and the code method of a program. It has taken not a short time to learn CUDA, and time and effort are large for application programmers.

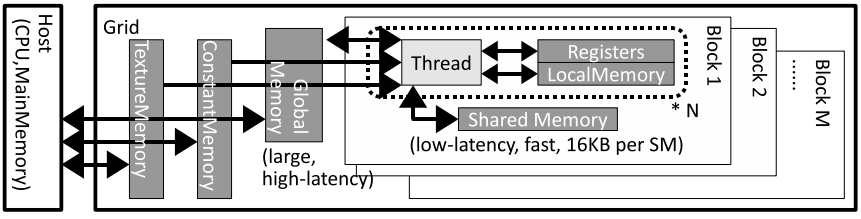


Fig. 3. Memory model of CUDA-capable GPU

3 Design and Implementation of OMPCUDA

The typical parallel program using OpenMP is a loop parallelization program using "for" specifier in C and "DO" specifier in Fortran. In SPEC OMP2001[4] which is a major benchmark suite of OpenMP, all 20 benchmark programs have "for" or "DO" specifier. Then, OMPCUDA is aimed at the typical loop parallelization program which used OpenMP. It effectively utilizes much parallelism in GPU.

3.1 CUDA and OpenMP

There are not a little difference between CUDA and OpenMP, so we have to propose how to assign OpenMP to CUDA. In this subsection, the idea of assignment in OMPCUDA is shown.

In OpenMP, the being of some threads and a shared memory is supposed, and threads carry out parallel processing using a shared memory. In CUDA, there are two parallel execution entities in particular CUDA Blocks and CUDA Threads, and there are hierarchical shared memories such as GlobalMemory and SharedMemory. So, we discussed the way to assign between CUDA and OpenMP as follows:

- Assign CUDA Threads to threads of OpenMP, and SharedMemory to shared memory (Fig. 4-2, Parallelization in MP).
- Assign CUDA Blocks to threads of OpenMP, and GlobalMemory to shared memory (Fig. 4-3, Parallelization in whole GPU).
- Assign CUDA Threads to threads of OpenMP, and GlobalMemory to shared memory (Fig. 4-4, Parallelization in whole GPU).

We aim at getting high performance in large size loop utilizing a highly parallelism of GPU. In this aim, it is necessary to utilize all CUDA Threads. In the shared memories of CUDA, a memory which can be read and write accessed by multiple CUDA Blocks is only GlobalMemory. Therefore OMPCUDA treat CUDA Threads as threads of OpenMP and GlobalMemory as a shared memory of OpenMP (Fig. 4-4).

In the parallel execution using OpenMP, only the portions which specified by application programmers are executed in parallel. Other portions are executed in

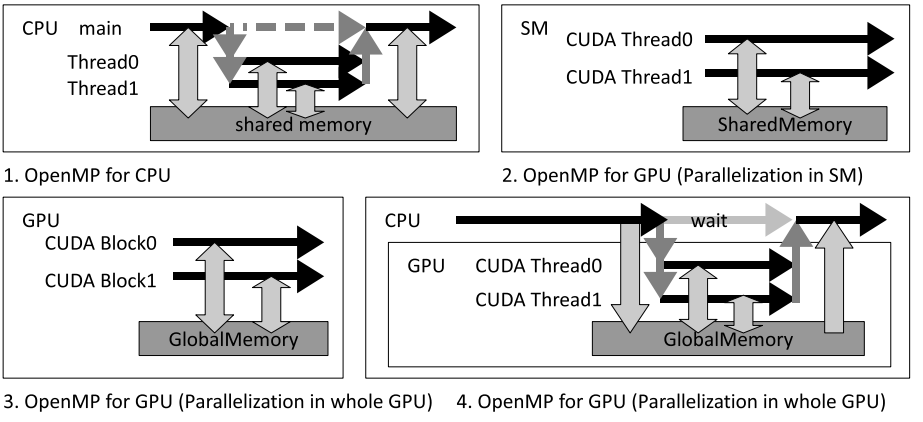


Fig. 4. Matching with parallelization programming and CUDA using OpenMP

serial. Therefore, the boundaries between serial execution portions and parallel execution portions are clear. On the other hand, the CUDA's parallel processing on GPU is also clearly distinguished from the function executed on CPU. It is possible to choose the portion in which parallel execution from the whole programs and to carry out parallel execution on GPU.

In CUDA, CPU and GPU have independent memories. It is necessary to send and receive data. On the other hand, serial execution portions and parallel execution portions are not performed simultaneously in the execution model of OpenMP. So correct result can be obtained by transfer all data on the boundary of a serial execution portions and a parallel execution portions.

Fig. 5 shows the OMPCUDA's assignment of OpenMP and GPGPU described above.

3.2 Implementation of OMPCUDA

We are developing OMPCUDA following the descriptions in subsection 3.1. In this subsection, an implementation of OMPCUDA is described.

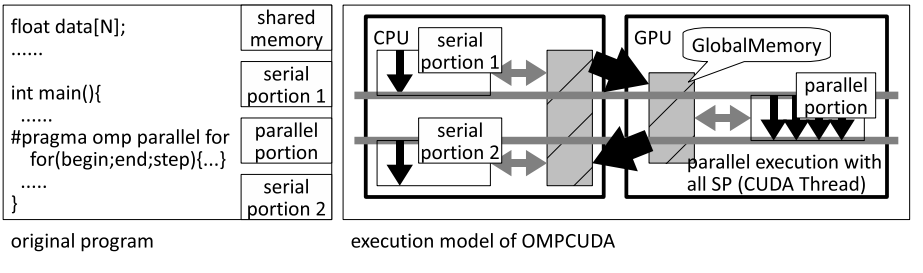


Fig. 5. Assignment of OpenMP to GPGPU

Implementation overview. Implementation of OMPCUDA is performed using OMNI version 1.6a. Since OMNI has multiple language front-ends, a toolkit, and a runtime library described above, we decided to implement OMPCUDA by using these features of OMNI. This implementation can reduce the burden of making a processor performing OpenMP specifiers.

In OMNI, the parallel execution portions are separated as a function and assigned to threads by the parallel execution start function of the runtime library. And then the runtime library provides inner-thread special functions such as a loop scheduling, a reduction, and a barrier synchronization. There are no memory copy operations because threads have a common shared memory.

On the other hand, the parallel processing on GPU with CUDA uses functions as the execution unit. By calling from CPU, the same programs are simultaneously executed with many computing elements on GPU. Data required in parallel portions are transferred to GPU from CPU before calling a Grid. After the Grid execution ends, the result is transferred from GPU to CPU.

There we developed a transformation mechanism and a runtime library. The relation between OMNI and OMPCUDA and the whole image of OMPCUDA are shown in Fig. 6. And the relation between a normal OpenMP program and the OMPCUDA's output source code are shown in Fig. 7.

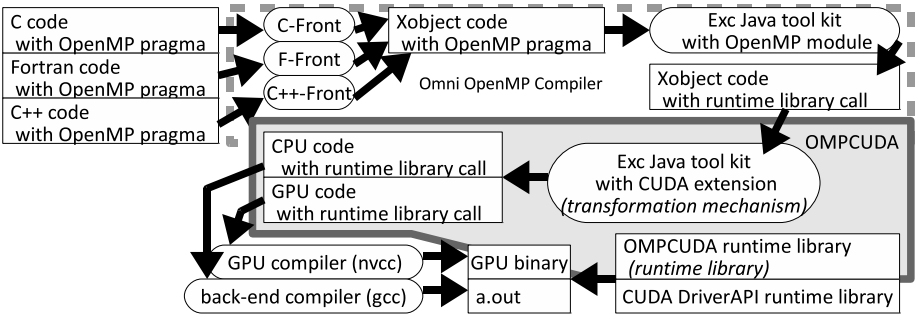


Fig. 6. Relation between OMNI and OMPCUDA, and an outline of OMPCUDA

Implementation of the program transformation mechanism. The program transformation mechanism of OMPCUDA rewrites parallel execution functions of OMNI to OMPCUDA specific procedures which include data transfer from CPU to GPU, the instruction execution start directions to GPU, and data transfer from GPU to CPU. For this purpose OMPCUDA must know what variables have to be transferred between CPU and GPU. Then the OMPCUDA's program transformation mechanism analyzes the program and detects the variables which are used in both CPU and GPU. And then, the mechanism rewrites runtime library calls in parallel execution portions to the OMPCUDA library calls.

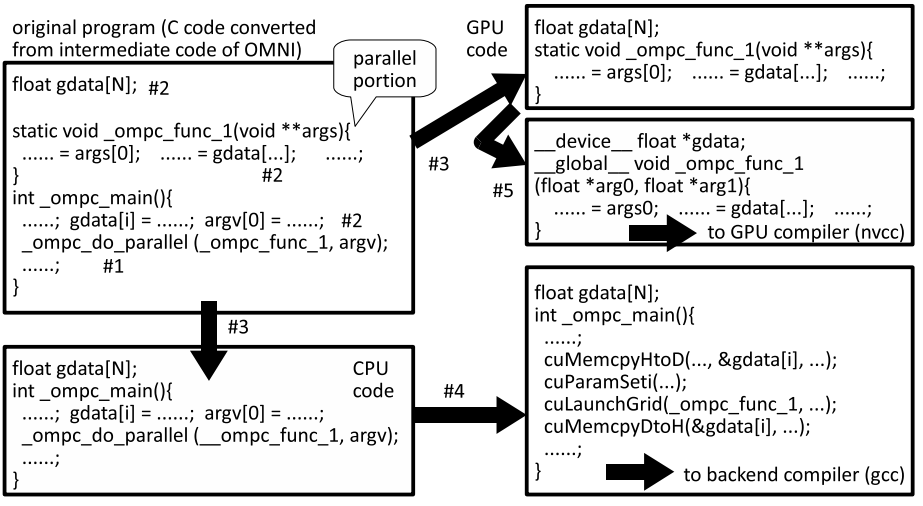


Fig. 7. Procedure of a program transformation mechanism

The procedure of transformation is follows:

1. From the whole program written with intermediate code, the start function of a runtime library is discovered. (Fig. 7-#1)
2. The variables used inside the parallel execution portions are validated. If these variables are not function local variables, they are considered as the variables which are needed to be transferred between CPU and GPU. (Fig. 7-#2)
3. Parallel execution portions are written out to other files, and they are deleted from the original source code. (Fig. 7-#3)
4. About the CPU code, the parallel execution start function call of OMNI is rewritten to the execution start function and the data transfer procedures. (Fig. 7-#4)
5. In accordance with the notation of CUDA, CUDA specifiers (__global__, etc.) are added to the GPU code at functions and variables. (Fig. 7-#5)

Implementation of the runtime library. The runtime library of OMNI offers thread creations, thread assignments, scheduling of parallel loops, functions of a reduction, and a synchronous processing between threads. In these, thread creations and thread assignments are performed by the translating mechanism. So the OMPCUDA's runtime library does remaining functions.

Scheduling of parallel loops. In OMNI, some scheduling methods in loop parallelization are offered. Application programmers can choose one of the methods using an OpenMP specifier. Only the simplest static scheduling is currently offered in OMPCUDA. This scheduling evenly divides the target parallel loop and simply assigns to CUDA Threads at runtime like as OMNI. An example is shown in Fig. 8.

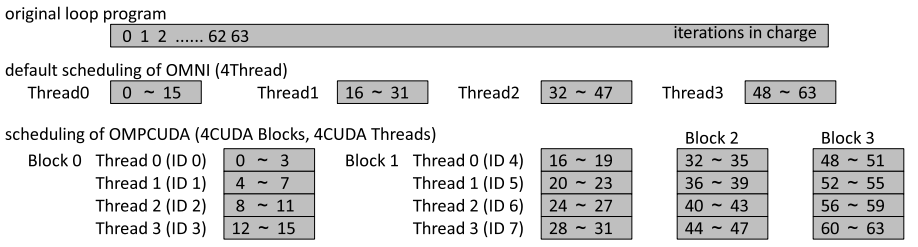


Fig. 8. Example of a scheduling of the parallel loop in OMPCUDA

In OMPCUDA, the usage number of CUDA Threads is 256 by default. 256 is the enough number which is said that CUDA can get high performance. The usage number of CUDA Blocks is the number of the parallel loop divided by the number of CUDA Threads. OMPCUDA experimentally has a changing mechanism of these sizes by environment variables and API functions like as OMNI.

Reduction. A reduction specifier is used in order to collect the variables which all the threads have at the end of parallel execution. There are already some fast reduction implementations using GPU or CUDA [5,6]. In OMPCUDA, the parallel reduction function which uses SharedMemory was implemented based on the technique of Owens[7].

After performing reduction for every CUDA Block, CUDA Thread in each CUDA Block has to perform final reduction. The simple and clearly way to this reduction is using atomic function. In this way, if the number of CUDA Blocks increases, the reduction time between CUDA Blocks will increase. Then we implement another reduction function in which all of CUDA Blocks returns the partial reduction data to CPU and CPU performs final reduction. This function increases the data size which transferred between CPU and GPU, but there are potentially to reduce execution time by reducing the number of atomic functions.

Other implementations. Because OMPCUDA uses Xobject code, it is not necessary to change the front-ends. But a few changes are needed to implement OMPCUDA smoothly.

It is necessary to execute the initialization and cleanup of CUDA and OMPCUDA, so the initialization and cleanup procedures are inserted into the real main function of OMNI.

4 Evaluation of OMPCUDA

In this section, OMPCUDA is evaluated as to whether the effect of parallel execution is acquired. Evaluation environment is shown in Table 1.

Table 1. Evaluation environment

CPU	Intel Xeon E5345 (4 cores, 2.33GHz)
Main memory	4.0GB
GPU	NVIDIA GeForce GTX 280 (240 SPs, core 602MHz / SP 1296MHz)
Video memory	1.0GB
connection	PCI-Express x16 (Gen 2)
OS	CentOS 5.0 (kernel 2.6.18)
Compiler	GCC 3.4.6, nvcc 2.0 v0.2.1221, Omni OpenMP Compiler 1.6

4.1 Performance Evaluation Using Matrix Product

A matrix product is a problem which has much computational complexity to the amount of data and the parallelism is high. It is widely known as a problem which can carry out high speed execution using GPU. A length of one side of a matrix will be called problem size. Comparison about each following implementation was performed in evaluation.

1. OMPCUDA-loop3: Implementation of the matrix product by simple 3-fold loop using OMPCUDA.
2. OMPCUDA-loop2: Implementation which two loops of the outside in OMPCUDA-loop3 were merged.
3. SimpleCUDA: Implementation of CUDA, SharedMemory is not used.
4. UseSharedCUDA: Implementation of CUDA, SharedMemory is used (the sample MatrixMul of CUDASDK2.0 itself).
5. OmniCPU-loop3: Loop parallelization using original OMNI, executed on only CPU, one thread and four threads (OmniCPU-loop3-1 uses one thread, OmniCPU-loop3-4 uses four threads, same source code with OMPCUDA-loop3).
6. AtlasCPU-nothread, AtlasCPU-pthread: CPU execution using ATLAS, executed on only CPU (one thread and four threads).

Fig. 9 shows the execution time of each implementation. The problem size is 1024, and the number of a CUDA Threads number of OMPCUDA is 256.

OMPCUDA-loop3 and OMPCUDA-loop2 differs in the performance greatly because the number of parallelism of OMPCUDA-loop3 is 1024/256=4, and it is 1024*1024/256=4096 of OMPCUDA-loop2 at most. Because GeForce GTX 280 has 30 MPs and CUDA-capable GPU can obtain good performance with high parallelism more than physical parallelism, so OMPCUDA-loop3 has not enough parallelism and did not get enough performance. From the result of OMPCUDA-loop2, OMPCUDA could accelerate the program which utilized the high parallelism which GPU has.

OMPCUDA is sure slower than optimized implements of both CPU and GPU, but it is important result that OMPCUDA could get good performance with a very simple source code which is the same as existing OpenMP program. This is a meaningful result in our purposes and motivation,

The experiment above is performed in C, also Fortran77 can get similar trends.

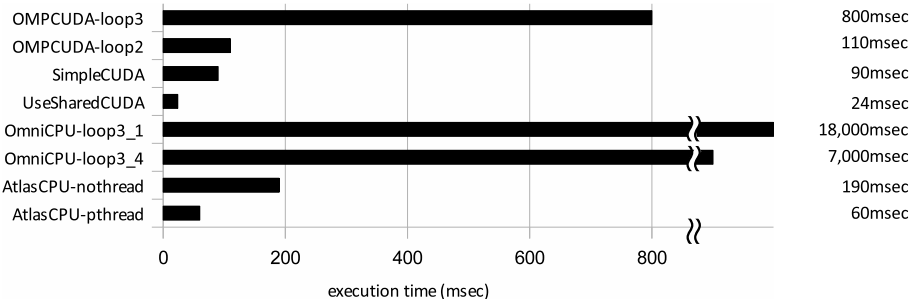


Fig. 9. Execution-time 1 of the matrix product (problem size 1024, single precision)

4.2 Evaluation Using pi Calculation (Gregory Series)

Gregory series is a program which is very simple and includes a reduction procedure. A reduction procedure is a very common-used procedure in real applications. The source code of the program used for evaluation is shown in Fig. 10.

```
float answer = 0.0f;
#pragma omp parallel for reduction(+:answer)
for(i=0;i<SIZE;i++){ answer += (4.0f / (4 * i + 1) - 4.0f / (4 * i + 3)); }
```

Fig. 10. Source code of the pi calculation

Comparison of the following implementation was performed in evaluation.

1. OMPCUDA: Implementation using OMPCUDA (Fig. 10).
2. SimpleCUDA: Implementation using CUDA, reductions are performed in GPU.
3. OMPCUDA-CPUreduction: Same as OMPCUDA except that CPU performs a part of reductions.
4. OMPCUDA-GLOBALreduction: Same as OMPCUDA except that GPU performs all reductions using atomic function.

The execution time of each implementation is shown in Fig. 11. When OMPCUDA is compared with SimpleCUDA, the difference of the execution time is only the time by rewriting of a loop. Execution time of CUDA-CPUreduction, which executes a part of reductions in CPU, is about 40% shorter. CUDA-GLOBALreduction is too slow.

Fig. 12 is the relative execution time of CUDA-CPUreduction when setting the execution time of OMPCUDA to 1. When there are a many number of CUDA Blocks, the difference has arisen. In the large size problem, OMPCUDA should use CPU in the portion of reductions.

This result hints that OMPCUDA can much more good performance by assigning the typical OpenMP procedures to the procedure of OMPCUDA runtime library.

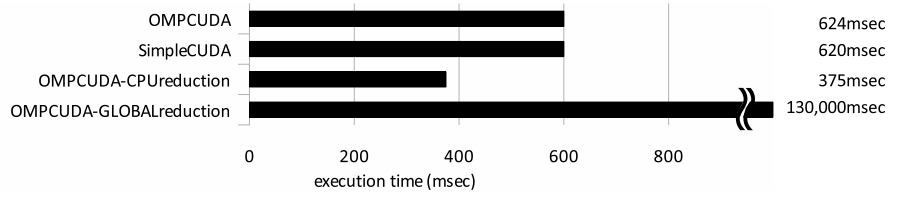


Fig. 11. Execution time of the pi calculation (problem size 0x10000000, single precision)

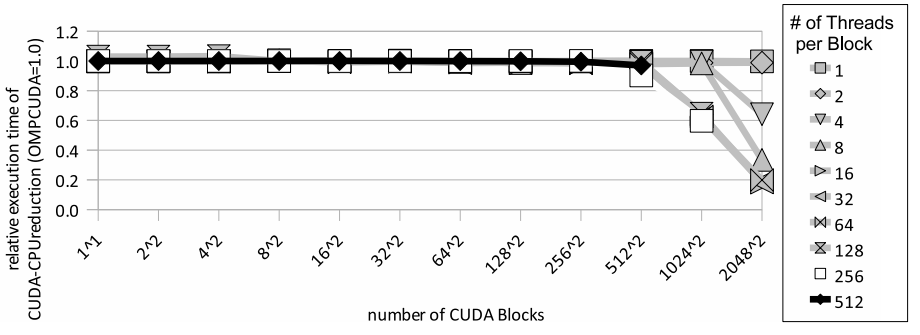


Fig. 12. Relative execution time of the pi calculation (problem size 0x10000000, single precision)

4.3 Experiment of SPEC OMP2001 Swim

We tried to execute SPEC OMP2001 swim (Fortran77 program, single source code) as a test of more realistic program. Swim is one of the smallest and simplest programs in SPEC OMP2001 benchmark, but it is a more realistic program than a matrix product and a pi calculation. Swim uses double precision floating point data.

As the result of execution, OMPCUDA could execute swim correctly, but took a large amount of time. The execution time of CPU using small data set named "test" with single thread is around 0.2 sec. The other hand, the execution time of OMPCUDA with multi Threads and Blocks using "test" data set is around 20 sec. The data transfer between CPU and GPU and the data assignment are the main reason. Swim program has large global array variables which are used in more than one parallel region (Fig. 13). In a current implementation, OMPCUDA transfers all the variables which are used in parallel portions when the beginning and the end of parallel portions. So OMPCUDA took a long time in transfer the variables. Current GPUs sure have low double precision performance (one eight of single precision), but this is not a critical reason.

To shorten the execution time, OMPCUDA has to reduce the time of data transfer. If the structure of target program is simple, the transfer time may be shortened by merging parallel regions. But swim has a determination of the end of benchmark between parallel portions, not a small-sized program changing

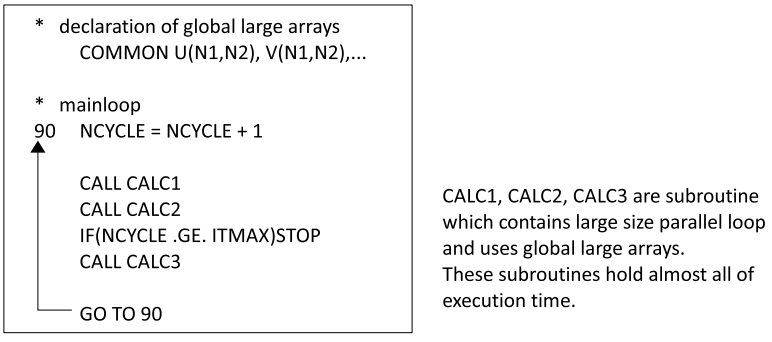


Fig. 13. Outline of swim benchmark program

is needed to shorten the execution time. If sequential portions between parallel portions don't use the large array variables, it will cause a performance improvement by omitting transfer and leave the array variables on GPU. But OMPCUDA now does not have such analysis features and OpenMP specification also does not have such data assignment specifiers.

It is important that OMPCUDA can execute OpenMP existing source codes without rewriting but it is not always true that good performance will be gotten with the same source codes as OpenMP program for CPUs.

5 Related Work

As mentioned before, CUDA programming is difficult and complex. Some studies which make CUDA programming easy are conducted. There are some compilers and execution frameworks which can make GPU program using pragma based languages such as OpenMP.

Lee et al.[8] are developing OpenMP compiler for CUDA. The compiler converts from OpenMP Input program to GPU specific Optimized OpenMP programs and converts from Optimized OpenMP programs to CUDA GPU programs. It has OpenMP-level and CUDA-level optimization mechanisms and has obtained high performance in some programs. Compared with its implementation, our OMPCUDA does not have good optimization mechanisms. But we will able to get such good performance by bringing in their optimization techniques.

The latest PGI compiler supports pragma-based parallel programming for CUDA in C/C++/Fortran languages [9]. But the language specification of pragma is PGI's own specification and not equal to OpenMP. Although we aim at getting good performance with OpenMP's standard pragma, we should discuss whether we should bring in non-standard pragma if the only OpenMP's non-standard pragma can describe something information which greatly affects performance.

Compared with their studies, it is OMPCUDA's specialty that OMPCUDA supports OpenMP on existing multiple programming languages and based on OMNI which is a known OpenMP compiler for CPU.

6 Conclusion

In this paper, we introduced OMPCUDA which is an OpenMP framework for CUDA based on Omni OpenMP Compiler. The choice of using OMNI has been a good aspect for omitting the burden of implementations which is similar with an OpenMP for CPU. In OMPCUDA, we assigned OpenMP to CUDA Threads and GlobalMemory of CUDA by focusing the high parallelism of GPU. As the result, we have gotten a good performance as planned in test programs. Moreover, while we assigned parallel portions to GPU basically, we got better performance by moving a part of runtime library from a parallel portion of GPU to a serial portion of CPU in a reduction procedure.

OMPCUDA is under development and has room for improvement. For example, moving data from GlobalMemory to SharedMemory and register is an important issue for getting better performance. OMPCUDA now use CUDA Threads flatly across CUDA Blocks, while shared memory of OpenMP is accessed by all threads, Therefore using SharedMemory as a shared memory of CUDA Blocks may be difficult. Also, it is very difficult to fully reduce the access to GlobalMemory by using SharedMemory and register, but it will be possible to get more performance. In other instances, other OpenMP specifier may be considered. OMPCUDA now supported a few important OpenMP specifiers, but there are some specifiers which are often used. The "sections" parallelization may be suitable for parallelization of CUDA Blocks.

Acknowledgment. Authors thank Omni Compiler project for releasing OMNI. And authors thank referees give helpful comments.

References

1. NVIDIA: CUDA Zone, http://www.nvidia.com/object/cuda_home.html
2. Ohshima, S., Hirasawa, S., Honda, H.: Proposal of GPGPU Programming Using Existing Parallelizing Method. IPSJ Tech. Report(ARC-175), 7–10 (2007)
3. Sato, M., Satoh, S., Kusano, K., Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster. In: EWOMP '99, pp. 32–39 (1999)
4. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
5. Horn, D.: Stream Reduction Operations for GPGPU Applications. In: GPU Gems2. Addison-Wesley, Reading (2005)
6. Roger, D., Assarsson, U., Holzschuch, N.: Efficient Stream Reduction on the GPU. In: Workshop on General Purpose Processing on Graphics Processing Units (2007)
7. Owens, J., Davis, U.: Data-parallel algorithms and data structures. In: SUPERCOMPUTING 2007 Tutorial: Hight Performance Computing with CUDA (2007)
8. Lee, S., Min, S.J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 101–110. ACM, New York (2009)
9. The Portland Group: PGI Accelerator Compilers, http://www.pgroup.com/resources/accel.htm

Author Index