Proceedings of the 6th European Workshop on OpenMP

EWOMP 2004

Proceedings of the 6th European Workshop on OpenMP

EWOMP 2004

Stockholm, Sweden
October 18 – 19, 2004

Sponsored by the OpenMP Architecture Review Board

Organising committee

Mats Brorsson, EWOMP'04 chair
Margreth Hellberg, administration
Sven Karlsson, proceedings

Program committee

Eduard Ayguadé, Univ. Politecnica de Catalunya, Spain
Mats Brorsson, KTH, Sweden
Mark Bull, University of Edinburgh, Scotland
Barbara Chapman, University of Houston, USA
Federico Massaioli, CASPUR, Italy
Timothy Mattson, Intel, USA
Dieter an Mey, Aachen University, Germany
Sanjiv Shah, Intel Americas Inc., USA
Matthijs van Waveren, Fujitsu, France
Per Öster, KTH, Sweden

Table of Contents

Session 3: OpenMP implementation issues

Chair: Dieter an Mey, Aachen University, Germany

Session 4: OpenMP tools 1

Chair: Per Öster, KTH, Sweden

Session 1: Performance evaluation

Chair: Mats Brorsson, KTH, Sweden

- H. Löf and S. Holmgren, affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System
- S. Kim, S. Hee, I. Park, and S. W. Kim, Performance Evaluation of SPEC OMP Benchmarks on the InfiniBand-Based Cluster System
- S. Sbaraglia, K. Ekanadham, S. Crea, and S. Seelam, pSigma: An Infrastructure for Parallel Application Performance Analysis using Symbolic Specifications

affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System

Henrik Löf and Sverker Holmgren
Uppsala University
Department of Information Technology
Box 337, 752 37 Uppsala, SWEDEN
{henrik.lof,sverker.holmgren}@it.uu.se

Abstract

To achieve close to optimal performance on cc-NUMA systems for shared memory parallel applications with complex data access patterns, a mechanism for co-locating threads and the data during the execution of the program is needed. The affinity-on-next-touch *procedure studied in this paper is based on re-doing the standard first-touch allocation at explicitly given locations in the code. We study the performance of a parallelized scientific computing application for which thread-data affinity can not be created by standard methods. We observe a performance improvement of 64% if the affinity-on-next-touch procedure is invoked once at a critical location in the code. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed the handling of page entries in the TLBs. The cost for actually migrating data is negligible. Using larger but fewer pages we measure a performance improvement of 127% compared to the original code.*

1. Introduction

The memory access patterns of scientific computing codes are becoming more and more complex. Unstructured grids and adaptive schemes are introduced in the computational models to increase the accuracy and resolution, resulting in that the data access patterns are more irregular and possibly also changes during execution. When parallelizing such codes, a shared memory programming model is preferred. One reason is that the programmer does not have to explicitly manage the communication between processors. Another important reason is that programming tools like OpenMP use a uniform shared memory model, implying that the programmer does not have to specify how the data should be partitioned and where it should be stored in the memory. However, modern scalable shared memory computers are normally of cc-NUMA type. In such systems, memory and CPUs are distributed over several nodes, where each node normally consists of a physical CPU-memory board. The access time for memory within a node is smaller than the time required for a remote memory access, and the difference is measured by the NUMA-ratio,

$$\text{NUMA-ratio} = \frac{\text{Remote access time}}{\text{Local access time}}.$$

On a NUMA system, the distribution of data in the shared memory has an effect on the performance of programs. Misplacement of data may cause that the performance and/or scalability is reduced even if the code is highly optimized in other aspects.

Possible ways of reducing negative performance effects arising from the mismatch between the programming model and the underlying computer architecture has been discussed by many authors, see e.g. [2, 13, 5, 9, 12]. The proposals fall into two main categories:

Extend the programming model: Here, the programmer may explicitly specify the data distribution using directives that are added to the programming model, see e.g [1]

Extend the computer system: Here, the OpenMP runtime system, the operating system and/or special hardware provide mechanisms for detecting and correcting misplacement of data in a transparent way while the programming model is left unmodified [16, 14, 13, 3].

Both of these approaches have advantages and disadvantages. If the programmer knows how data is accessed, it may be easy to explicitly specify a good data distribution. Then, a programming model where this distribution can be described directly will probably result in good performance. On the other hand, if the data access pattern is complex, a

substantial effort may be needed by the programmer to determine a good data distribution, or it may not even be possible. In such situations, a solution where the computer system adaptively determines the data distribution in a transparent way may be more beneficial. However, there is a risk that the overhead introduced by the modifications of the system will result in a loss of performance and/or scalability, counteracting positive effects of improving the data distribution.

In this paper, we study a method for dealing with the data distribution problem based on a run-time mechanism for co-locating data and the threads that use it on the same node of the computer system. The mechanism can be made available to the programmer by a portable and simple addition to the programming model. We evaluate the performance effect of applying this method to an industrial scientific application, parallelized using OpenMP and executed on a cc-NUMA computer. The results show that it is possible to decrease the execution time significantly compared to if the original code is used. The results also show that, on the computer system studied, the overhead introduced by the directive is dominated by handling TLB coherence and not by the actual movement of data. This implies that the overhead can be reduced by using large page sizes.

2. Thread-data affinity on cc-NUMA systems

When executing a parallel program on a cc-NUMA system, it must be determined how the data should be partitioned over the nodes and at which nodes the threads should be run. To get high performance, the number of accesses to remote memory should be small. This can be achieved if the *thread-data affinity* is large, i.e. if each data item resides in the memory of the node where the thread that uses it the most is running.

The standard method for creating thread-data affinity is to use a first-touch strategy. When a memory page is accessed for the first time during execution, the operating system has to reserve a physical page for it in one of the nodes. Using first-touch data placement, the page is allocated in the node where the thread responsible for the first access is running. The hope is that this thread will also be the one that accesses the page the most later, and that the thread will continue to run in the node where it was started.

If the data access pattern changes after the initialization or if the threads move between nodes during execution, the thread-data affinity is normally reduced. The first situation occurs for example if the data is initialized by a single thread before the parallel region in the program begins. In such cases the first-touch strategy normally results in that all data is allocated on a single node. This problem might sometimes be tackled by rewriting the code so that it performs a parallel initialization of the data, see e.g. [10, 9]. However, in other cases a substantial programming effort is needed to achieve this. Furthermore, some form of indirect addressing is used in many codes, for example for handling unstructured computational grids. Then, the data access pattern is determined by the data itself, and it is not possible to exploit the standard first-touch strategy in an optimal way. In such cases, a viable approach for creating thread-data affinity is to introduce a mechanism for re-doing the first-touch placement of data at some locations in the code during the execution. This would allow a re-creation of thread-data affinity by effectively migrating data to the correct nodes. We use the term *affinity-on-next-touch* for such an operation.

If the access pattern of the main part of the program is static, it might be sufficient to invoke the affinity-on-next-touch procedure once after the initialization. For example, for an application using an unstructured grid, the procedure could be invoked for migrating data to the correct nodes once the access pattern is determined. In general scientific computing codes, a few distinct phases with different access patterns can often be identified. It is normally easy for the programmer to determine where affinity should be created, which implies that it is reasonable to add an affinity-on-next-touch directive to the programming model. A directive of this type is also mentioned in [14], and a first implementation is found in the Compaq OpenMP compilers [1]. An affinity-on-next-touch directive does not specify how and where to migrate data, which makes is portable and easy to use. Moreover, the time-consuming phases in computational codes normally consist of iterations where the data structures are repeatedly accessed using the same access pattern. In such cases, the overhead introduced by invoking the directive before the loop is amortized over the iterations.

If the data access pattern changes more often during the execution, the affinity-on-next-touch mechanism could be invoked repeatedly, or a fully transparent scheme for adaptive data distribution could be used. In a transparent scheme, ways of detecting candidate pages for migration and/or replication must be built into the system, and algorithms for deciding where and when to move data must be frequently executed. When implementing schemes of this type, great care must be taken not to introduce significant amounts of overhead.

3. The application

To evaluate the applicability of an affinity-on-next-touch directive, we study the performance of an industrial solver for the Maxwell equations in 3 dimensions, describing the scattering of electromagnetic waves from e.g. an airplane [7]. The solver uses an unstructured grid and a finite element discretization, and solves the resulting sparse system

of equations using a conjugate gradient (CG) solver. In the iterative solver, a bandwidth minimization algorithm [8] has been introduced [11]. The code is written in Fortran 90 and C, and parallelized using OpenMP.

The data access pattern in the computations is static and unstructured, and it is possible identified two main phases in the code:

Assembly of iteration matrix This part of the code is not very time-consuming. Also, it is rather complex and hence not parallelized

Iterative solver This computationally heavy part of the code is parallelized using an algorithm with a small number of synchronization points [11]. For normal application problems, about 50 iterations are required to compute the solution

In the experiments presented below we study a problem where the iteration matrix has 1794058 rows, and each row has on average 15 non-zeros entries. This leads to that the size of the data set is approximately 500 MByte.

Since the assembly of the iteration matrix is performed by a single thread, all of the data will be allocated to a single node after the assembly phase. By invoking the affinity-on-next-touch procedure just before the first iteration in the iterative solver, the data will be migrated to several nodes matching the location of the multiple threads used in the parallel region. In our experiments, we run the iterative solver to convergence once, and investigate the performance gain from creating affinity for this computation. In a real-life computation, the system of equations is solved over and over again for hundreds of time steps, implying that the overhead associated with the creation of thread-data affinity will be amortized over many more iterations.

4. The cc-NUMA system

The experiments are performed on a dedicated 32 CPU domain on a Sun Fire 15000 system (SF15K) [4]. Here, each node consists of four 900MHz UltraSPARC-IIICu CPUs, each having 8 MByte of L2 cache and 4 GByte of main memory. The system runs Solaris 9, where the affinity-on-next-touch mechanism can be implemented by a call to the `madvise(3C)` library routine.

In the experiments, the setting of a kernel parameter for the Solaris scheduler is modified to increase the probability that the threads stay on the same nodes during the execution of the program [15]. We did not use any binding to CPUs or processor sets.

It should be noted that NUMA-ratio of the SF15K system is only approximately two. This means that we can not expect that the effects of modifying the distribution of data to be dramatic.

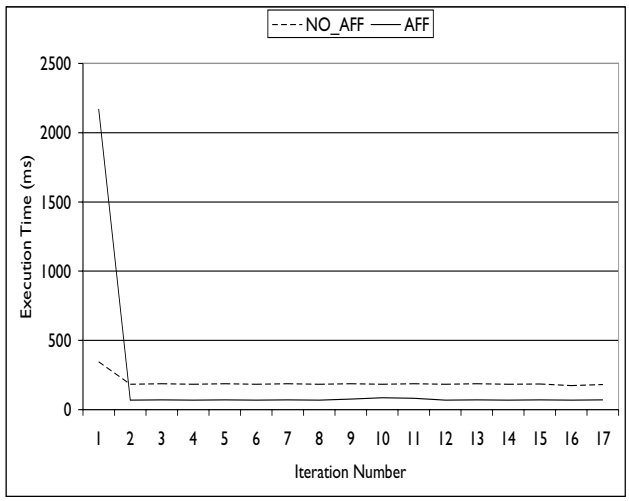


Figure 1. The effect of the affinity-on-next-touch procedure when 16 threads are used. The directive was executed just before the first iteration. Only the first 17 out of 47 iterations are shown.

5. Results

In Figure 1, the effect of invoking the affinity-on-next-touch mechanism is illustrated. In this experiment, 16 threads are used. The graph shows the execution time for each iteration if the original code is used (`NO_AFF`) and if the affinity-on-next-touch mechanism is introduced before the first iteration in the solver (`AFF`).

In the `NO_AFF` case, the data remains allocated on a single node. The number of remote accesses is large, resulting in a rather large execution time per iteration and a total execution time of 9.35 s. The slightly larger execution time for the first iteration can be attributed to cold-start effects, where some of the data is cached on multiple processors when the parallel region is entered.

For the `AFF` case, it is clear that an affinity-on-next-touch directive affects the performance in two ways. In the first iteration, there is a cost for creating the affinity and the execution time for this iteration is increased. After this, the execution time per iteration is significantly reduced. Using the UltraSparc-III hardware counters, a measurement of the number of L2 cache misses that are served from remote memory verifies that the data is actually migrated to the nodes involved in the parallel computations. Therefore, we conclude that the thread-data affinity actually has a positive effect on performance. The average execution time for iteration $2 - 47$ is reduced from 185 to 71 ms, and the total execution time is reduced to 5.91 s.

In a more general situation, it is clear that the initial cost

of creating affinity must be amortized over a number of iterations (or similar) if the total performance should be increased. A simple calculation shows that for our experiment, 17 iterations are needed before the break-even point is reached.

It is interesting to study of the overhead for the affinity-on-next-touch procedure in more detail. In the experiment presented in Figure 1, we know that data is actually migrated from one node to several others at the first iteration. The overhead for this can be divided into two parts: The cost of copying data and the cost of keeping the page tables coherent. The copy operations are parallelizable, and the performance is only limited by the memory system characteristics. However, the modifications of the page tables are potentially more more problematic.

If a page translation is cached in several translation lookaside buffers (TLBs), we need to enforce coherency among these buffers. This procedure is serial, and involves an expensive *TLB shoot-down* process where dirty translations in the TLBs of processors sharing the page are replaced. This process often involves global kernel locks, polling and/or interrupts [6]. To investigate the relative effect of the two types of overhead, another experiment was performed where the affinity-on-next-touch procedure is invoked twice, once at iteration 15 and once at iteration 40. Since the access pattern is static during the iterations, no data is actually moved on the second invocation. The result show that there was no measurable time-difference iterations 15 and 40, and it is clear that the overhead is dominated by handling the TLBs. This assumption is further supported by results from measuring the number of minor page faults and cross-processor interrupts using kernel statistics.

As described above, we have found that the overhead connected to creating thread-data affinity is dominated by handling entries in the TLBs. It is reasonable to assume that using fewer but larger pages could reduce this overhead. In Figure 2, the effect of using 64kB pages instead of the standard 8kB pages is show, again for an experiment using 16 threads. In the figure, the two curves in Figure 1 are repeated (`NO_AFF8kB` and `AFF8kB`), but now together with the corresponding curves for 64kB pages (`NO_AFF64kB` and `AFF64kB`). It is clear that the overhead in the first iteration is reduced by a factor of 3 when using 64kB pages compared to the 8kB case. Also, the performance for the subsequent iterations is almost unaffected by changing the page size. The total execution time is now reduced to 3.9 s, and the break-even point where the overhead has been amortized is reached already after four iterations. The primary reason for the reduction of the overhead when using a larger page size is that fewer TLB entries must be handled. In the TLB shoot-down process, the initiating thread sends one interrupt per page to invalidate possible shared TLB entries for other threads. The UltraSparc-III processor does not have hardware support for 64kB pages, implying that when 64 kB pages are used, in practice one interrupt invalidates eight 8kB pages. The results for the original code in Figure 2 indicate that there is another, smaller positive effect provided by a general reduction of the cold-start effects for entering the parallel region when the number of TLB entries needed to cache the working set is reduced.

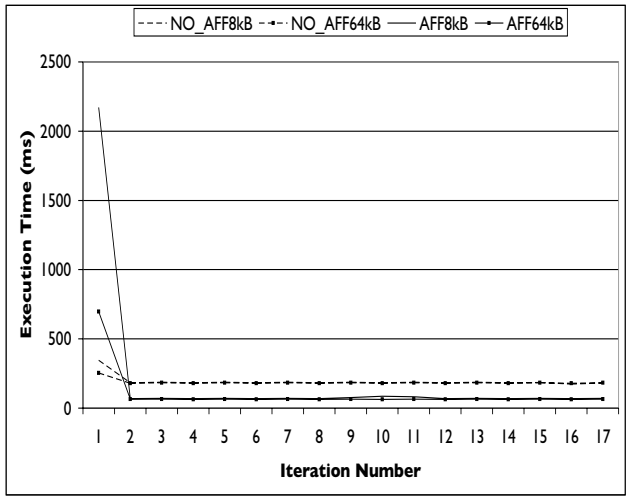


Figure 2. A comparison of the effect of the affinity-on-next-touch procedure when 8kB and 64kB page are used.

In Figure 3, the effect of invoking the affinity-on-next-touch procedure for different numbers of threads is shown. The bars show the improvement of the total execution time when using 8kB and 64kB pages compared to the cases where the original code is executed using 8kB pages using the same number of threads. From the figure, it is clear that enforcing a proper distribution of data by invoking an affinity-on-next-touch procedure in general improves the performance for the application studied here. Also, it is clear that in all cases the performance improvement is larger when 64kB pages are used instead of 8kB pages. The performance decrease for 2 threads and 8kB pages is explained by that in this case the two threads are scheduled to the same node, and there will be no possible benefit of migrating data. Then, the overhead in the first iteration is not followed by any performance improvement in subsequent iterations.

A more detailed study of the experiments described by Figure 3 reveals that, using the implementation studied in this paper, the affinity-on-next-touch mechanism is not scalable. The initial iteration overhead is almost constant as the number of threads is increased. As the iterations become faster when more threads are used, more and more iterations must be performed before the break-even point is reached

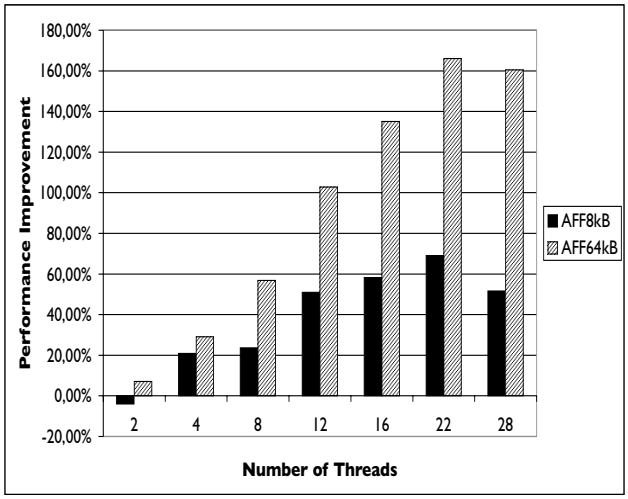


Figure 3. Performance improvement of using the affinity-on-next-touch procedure compared to if the original code is used. Results for both 8kB and 64kB pages are shown.

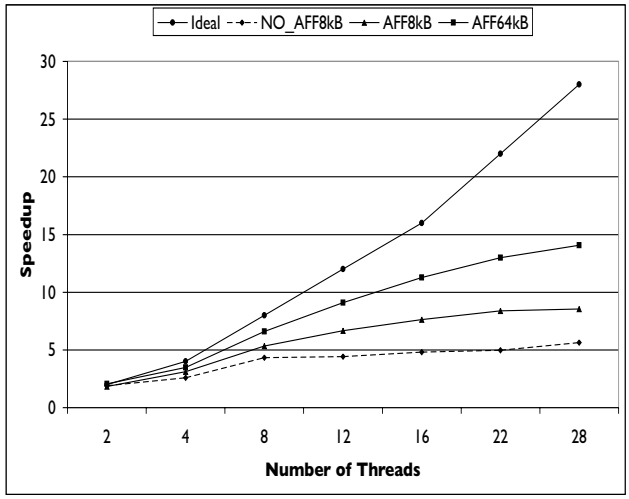


Figure 4. The speedup of the solver using the original code (`NO_AFF8kB`), affinity-on-next-touch and 8k pages (`AFF8kB`), and affinity-on-next-touch and 64k pages (`AFF64kB`).

and the initial overhead is amortized. However, this effect is reduced by using 64kB pages.

Finally, we present a traditional speedup graph where the code is executed using different numbers of threads and the affinity-on-next-touch procedure is invoked as described earlier. Results for both 8kB and 64kB pages are shown. 4. From the figure, it is obvious that reducing the initial iteration overhead has a positive effect on the scalability of the parallel implementation.

6. Conclusions

To achieve close to optimal performance on cc-NUMA systems for shared memory applications with complex and possibly changing data access patterns, a mechanism for creating or re-creating thread-data affinity during the execution of the program is needed. In this paper, we have evaluated such a procedure, denoted *affinity-on-next-touch*, based on re-doing the standard first-touch allocation at given locations in the code. This normally results in that data is migrated between the nodes in the system. The experiments are performed using a parallelized scientific computing application where thread-data affinity can not be created by standard methods. We find that the execution time can be significantly reduced by invoking the affinity-on-next-touch procedure. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed the handling of page entries in the TLBs. The cost for actually migrating data is negligible. Led by this observation, we manage to reduce the overhead by increasing the page size, which results in that fewer TLB entries must be handled.

Since the affinity-on-next-touch procedure is invoked at distinct locations in the algorithm, it is natural to suggest that it should be accessible by a directive in programming tools like OpenMP. The programmer usually knows where thread-data affinity is critical for performance. In OpenMP, other options are to add an affinity-creation attribute to the parallel region directive, or to automatically invoke the procedure for creating affinity at the beginning of each parallel region. The latter solution has the advantage that the programming model is left unmodified.

References

[1] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA machines. *Scientific Programming*, 8:163–181, 2000.

[2] T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, Sept 1993.

[3] J. M. Bull and C. Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP*. `http://www.caspur.it/ewomp2002/`, 2002.

[4] A. Charlesworth. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press, 2001.

[5] J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129. ACM Press, 2003.

[6] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.

[7] F. Edelvik. *Hybrid Solvers for the Maxwell Equations in Time-Domain*. Doctoral thesis, Mathematics and Computer Science, Department of Information Technology, University of Uppsala, may 2002. `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-2156`.

[8] N. E. Gibbs, J. William G. Poole, and P. K. Stockmeyer. An Algorithm for Reducing the Bandwith and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.

[9] S. Holmgren, M. Nordén, J. Rantakokko, and D. Wallin. Performance of PDE Solvers on a Self-Optimizing NUMA Architecture. *Parallel Algorithms and Applications*, 17(4):285–299, 2002.

[10] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[11] H. Löf and J. Rantakokko. Algorithmic Optimizations of a Parallel Industrial CEM Solver. Technical report, Dept. of Information Technology, Uppsala University, 2004.

[12] T. G. Mattson. How good is OpenMP. *Scientific Programming*, 11:81–93, 2003.

[13] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8:143–162, 2000.

[14] L. Noordergraaf and R. van der Pas. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38. ACM Press, 1999.

[15] Sun Microsystems, `http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf`. *Solaris Memory Placement Optimization and Sun Fire servers*, January 2003.

[16] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289. ACM Press, 1996.

Performance Evaluation of SPEC OMP Benchmarks on the InfiniBand-Based Cluster System*

Shinyoung Kim Seo Hee Inho Park Seon Wook Kim
Department of Electronics and Computer Engineering
Korea University, Seoul, Korea
http://compiler.korea.ac.kr

Abstract

Recently many advanced hardware-based interconnection schemes to connect physically distributed processing nodes have been developed to overcome performance limitation of networks, and one of them is the InfiniBand Architecture (IBA) which supports shared-memory programming semantics by means of remote direct-memory access (RDMA) and atomic operations. In this paper, we evaluate performance of SPEC OMP benchmarks on the InfiniBand-based Distributed Virtual Shared-Memory (DVSM) system in detail. Our research helps the system and application developers to estimate the applicability of the OpenMP applications on a commodity cluster of workstations to use the state-of-the-art interconnection networks for the next decades.

1. Introduction

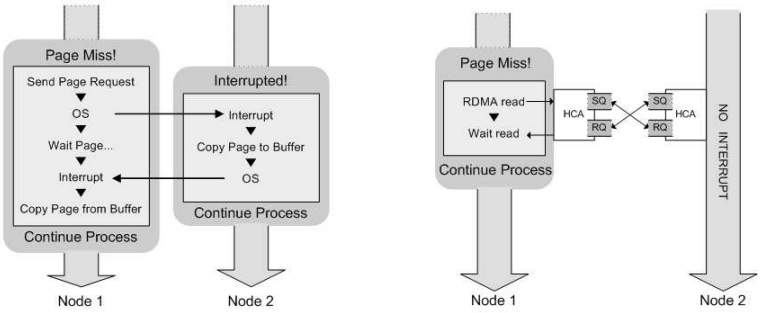
In order to provide easy programming environment on the distributed-memory system, many distributed virtual shared-memory (DVSM) systems have been introduced [1, 8, 15]. The DVSM systems allow processes to access physically distributed memory spaces through one software-enabled virtual shared memory space, and therefore a programmer is able to use the shared-memory parallel programming APIs on the system, such as OpenMP [13] and PThread. However, the performance of applications on the DVSM system, especially when executing parallel applications, heavily depends on the network performance and network programming semantics because data coherence mechanism across processes on the software-enabled virtual shared memory space is supported only by software over the network.

* This work was supported by the Electronics and Telecommunications Research Institute (ETRI), Taejeon, Korea.

In the previously designed DVSM systems [1, 8, 15] a process should communicate directly with other processes in order to exchange data through process interruption as shown in Figure 1 (a). Obviously this approach results in large execution overhead due to signal handling for process interruption and communication latency, and it degrades the total system performance significantly. Differently from the previously used schemes, on the InfiniBand-based DVSM system a process does not need to interrupt other processes in order to access data on remote nodes because the InfiniBand architecture provides hardware legacy software protocol tasks to support Remote Direct Memory Access (RDMA) and atomic operations (Figure 1 (b)).

The InfiniBand Trade Association [4] run by Intel, IBM, Dell Computer, and so on, has proposed the InfiniBand Architecture (IBA) in 1999, which consists of processing nodes, I/O nodes, and System Area Network (SAN) to connect nodes. Figure 2 gives an overview of the IBA. The distinguished features in the IBA are using a channel based interconnection switched fabric, and supporting Remote Direct Memory Access (RDMA) and atomic operations (i.e. compare and swap, fetch and add). The switched fabric in the IBA guarantees high stability and wider bandwidth. The IBA can be used in constructing from a small size cluster which has one processor and a few I/O device, to a large-scale parallel supercomputer consisting of hundreds of processors and thousands of I/O devices. The detailed description about the InfiniBand architecture can be found in [4].

In this paper, we evaluate performance of SPEC OMP benchmarks on our designed InfiniBand-based Distributed Virtual Shared-Memory (DVSM) system by comparing with the IPoIB-based traditional system on the IBA, especially TreadMarks [1]. Also we discuss code optimization issues. More specifically, the serializing small parallel code sections, which is the most effective scheme on the hardware SMP, is not applicable on the DVSM system and even makes the performance worse due to increment of data movement between serial and parallel sections. We argue that this paper contribute to help the



(a) `Without RDMA.` (b) `With RDMA.`

Figure 1. The difference between two schemes to access data on remote nodes.

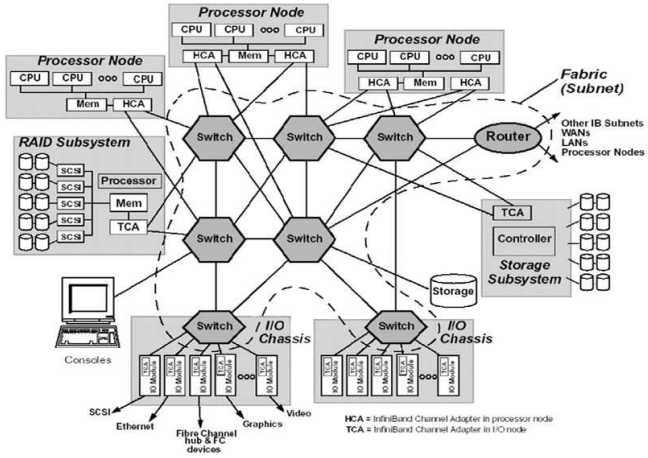


Figure 2. InfiniBand System Area Network [4].

system and application developers to estimate the applicability of OpenMP applications on a commodity cluster of workstations to use the state-of-the-art interconnection networks for the next decades.

The paper is organized as follows. Section 2 briefly presents our IBA-based DVSM implementation [14]. In Section 3 we characterize the performance of SPEC OMP benchmarks in detail, and in Section 4 the conclusion is made.

2. InfiniBand-Based Distributed Virtual Shared-Memory Systems

The data coherence mechanism across processes needs to maintain at least the following three information: modified data and their references per process, data access history (i.e. read and write) per coherence granularity, and access ordering (i.e. time). Our implementation is based on the lazy release consistency (LRC) model, which allows the protocol to aggregate the transmission of shared memory writes until a later synchronization point in order to minimize data communication between processes [1]. The synchronization between processes provides access ordering information, and *twin* and *diff* mechanisms distinguish between original and modified data, i.e. we know which references are modified [1]. In the TreadMarks system, a process to request data should interrupt a data providing process for communication, and therefore processes naturally know access history at communication time. But because the RDMA feature removes process interruption for data communication, we need a new mechanism to track access history on the InfiniBand-based DVSM system.

In order to record read and write operations per page (our system's granularity), we use an interval table whose entry has two bits to mark read and write operations and whose number of entries is equal to the number of pages in globally allocated virtual shared-memory. When a page segmentation violation occurs, the related bit is set. The interval table is cleared after all necessary diff operations are performed to other processes, since the event history is not needed any more (at this time dependences are completely resolved).

Each synchronization includes two internal barriers. At the first barrier each process broadcasts the interval table to all the participant processes, and at the second barrier the diff operations are applied to other processes if necessary by considering all the other processes' interval tables. It is well known that the overhead associated with the page segmentation violation is very large, and it is one of the most important factors to determine the overall performance. In order to reduce the occurrence of segmentation violations, we assign an ownership to each page. Basically the page owner is not memory-protected, and it is write memory-protected only after other processes perform read and write operations. The page owner information is maintained inside the interval table by extending one bit to mark a page owner-

ship. The details are shown in [14].

There are a few research activities about programming environment on top of the InfiniBand. The research in [6] describes the implementation of the OS-layered DVSM on the IBA. This approach needs to modify kernels in order to integrate the InfiniBand primitives with OS kernels. Our approach is the application-layered InfiniBand-based DVSM, and therefore we do not need any modification inside kernels. And while our framework uses a lazy release consistency, the DVSM in [6] uses a sequential consistency. Also the InfiniBand-based MPI implementation has been done extensively [9].

3. Performance Evaluation

3.1. Experiment Methods

For performance evaluation of SPEC OMP3.51 suite [3] on our IBA-based DVSM system, we used two versions of four benchmarks (`swim`, `mgrid`, `wupwise`, and `applu`). One version is to use officially delivered benchmarks from SPEC (called as unoptimized versions), and another is to parallelize only important loops (their execution time is more than 95% of the total execution in serial codes) in order to achieve the maximum performance by eliminating parallel code overhead (called as optimized versions) [2]. The parallel code overhead means code transformation overhead from OpenMP codes into thread-based forms by the OpenMP preprocessor and thread management overhead. We scaled down the number of iterations in SPEC OMP for short execution (it does not change any program characteristics) and we used `ref` data set.

We implemented the DVSM system on the InfiniBand to use Verbs API in order to use full features of the InfiniBand semantics. We used eight processing nodes connected with the 4X Mellanox InfiniBand, and each node includes a Intel 2.0GHz Xeon processor, 512MB memory, and 133MHz 4X PCI-X 128MB memory HCA board. Also for comparing with the performance on the hardware SMP machine, we measured the SPEC applications on the bus-based SMP to use four 2.8GHz Xeon processors and 512MB memory.

For execution of the OpenMP benchmarks on the DVSM system, we used the Polaris parallelizing compiler infrastructure [5]. The Polaris compiler translates the OpenMP directive codes into thread-based programs like a normal OpenMP preprocessor [12, 16], and links with runtime libraries to manage processes on the DVSM system.

There are the following six measurements for each benchmark: Execution of unoptimized versions on the hardware-based SMP machines (`UNOPT_SMP`), TreadMarks (Version 1.0.3.3) execution of unoptimized versions to use IPoIB protocol stack on the InfiniBand (`UNOPT_TMK`), and execution of unoptimized and optimized versions on our InfiniBand-based DVSM system (`UPOPT_IBA` and `OPT_IBA`).

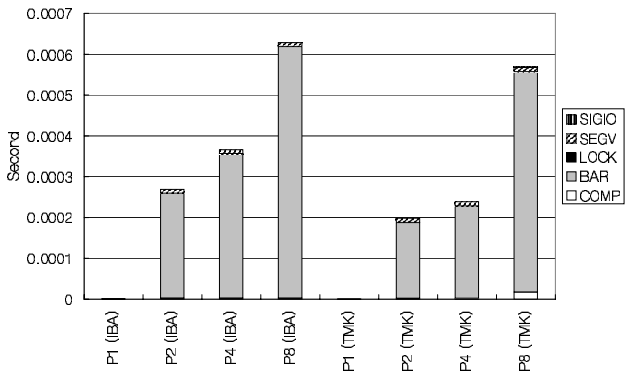


Figure 3. Process fork/join overhead of the OpenMP parallel directive on our system (`IBA`) and TreadMarks (`TMK`).

The execution time is categorized into the following performance metrics: `COMP` (execution time of the application itself), `BAR1` (overhead at the first barrier on the InfiniBand implementation or the total barrier time on the TreadMarks system), `BAR2` (overhead at the second barrier on the InfiniBand implementation), `BAR` (sum of `BAR1` and `BAR2`), `LOCK` (lock related overhead), `SEGV` (overhead related to page segmentation violation due to page misses), and `SIGIO` (overhead due to the process interruption from a service request process).

3.2. Performance

3.2.1. Overhead of OpenMP Primitives In order to measure overhead of OpenMP primitives such as `parallel`, `atomic`, and `shared`, we designed a microbenchmark. Figure 3 shows the process fork/join overhead, i.e. the overhead related to `parallel` OpenMP directive on our system and TreadMarks on one to eight processing nodes. The overhead in both measurements proportionally increases with the number of processes, and its value on our system is higher than TreadMarks. The reason of the performance difference is that as mentioned in the previous section, our system maintains an interval table to track access history, and at the synchronization point a process needs to broadcast the interval table to all the other processes for page diff actions. The table size per process is about 6.4KB for 100MB virtually allocated shared-memory space.

Figure 4 shows overhead of OpenMP `atomic` primitive on our system and TreadMarks. The lock overhead on the InfiniBand-based DVSM system is constant because the InfiniBand architecture provides hardware-implemented

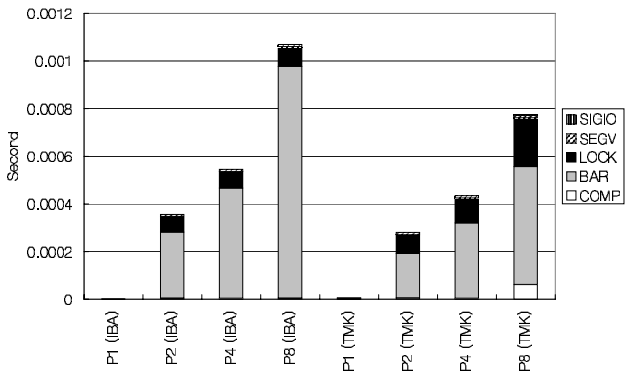


Figure 4. Overhead of OpenMP atomic primitive on our system (`IBA`) and TreadMarks (`TMK`).

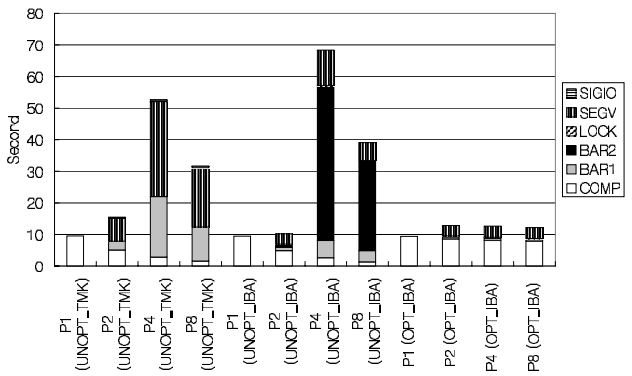


Figure 7. Overhead analysis in `APPLU`.

atomic semantics. On the TreadMarks system, the overhead increases proportionally with the number of processes.

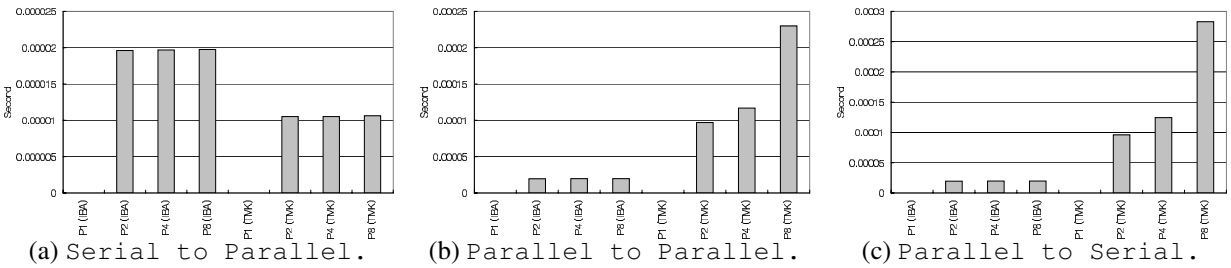
Figure 5 shows overhead of page movement between serial and parallel sections, i.e. related to `shared` data on our system and TreadMarks. In order to measure overhead only related to `shared` data movement we evaluated the overhead occurred by page segmentation violation. The overhead of data movement from parallel to serial and inside parallel sections on our system is much lower than that on TreadMarks, since our system uses RDMA features provided by the InfiniBand architecture. But from serial to parallel sections the overhead on our system is larger than on the TreadMarks system. Our system uses the home-based LRC protocol. In other words, a page owner is write-memory protected after child processes read the data from a master process, i.e. the page owner [7]. It incurs the segmentation violation overhead on a page owner. On the TreadMarks system multiple owners are allowed (homeless LRC protocol), and therefore there are less segmentation violations on a page owner process.

3.2.2. Overall Performance of OMP Benchmarks Figure 6 shows the speedup of the OpenMP benchmarks on 2, 4, and 8 processors with respect to one processor execution. In `applu` and `swim` there is no speedup in the DVSM executions, and in `mgrid` there is speedup at the execution of unoptimized version on our DVSM system, but less than the SMP execution. In `wupwise`, the speedup on both DVSM systems is higher than the SMP execution. The result in the figure is different from that in [11]. The reason of the performance difference is that our processor's performance is much higher than those in [11], so the communication overhead dominates the overall execution time.

Also the figure shows that the most effective optimization scheme on hardware SMP machines, i.e. serialization of small parallel code sections in order to reduce the parallelization overhead, is not effective on the DVSM system in all benchmarks except for `applu` (in `applu` we do not use the serialization and it will be discussed shortly). We found that the parallel code overhead is not important, and the most important consideration for performance optimization is data (page) locality. More specifically, the serialization of parallel sections incurs more data movement between serial and parallel sections, i.e. between host and child processes. It incurs more page segmentation violations when child processes start parallel region execution to use data updated in serialized parallel sections. It makes programmer optimize codes very difficult, and traditional optimization techniques used on the hardware SMP may be ineffective on the DVSM system.

3.2.3. APPLU Figure 7 shows that the unoptimized version of `APPLU` has enough parallelism, since `COMP` metrics are well scaled down proportionally with the number of processes. But in this code version, the overhead is very large because the most important loop `ssor_do_2_2` includes the barrier synchronization per iteration. Since we use the LRC protocol, the large number of barriers incurs huge diff operations and eventually degrades the overall performance significantly. The barrier overhead on our system is larger than on the TreadMarks system, and it was discussed in Figure 5 (a). When parallelizing only important loops [2] except for `ssor_do_2_2`, the overhead is significantly degraded. But there is no performance gain, because we parallelized only 18% of the serial code execution.

3.2.4. MGRID Figure 8 shows execution overhead in `MGRID`. There is a little speedup in unoptimized versions, but the performance is significantly degraded when only parallelizing important loops due to excessive page segmentation violations in serialized `zero30_do_1`, `zran3_do_1`, and `zero3_do_1` code sections. On the hardware SMP machines, we have serialized small parallel sections in order to remove parallelization overhead. But



(a) `Serial to Parallel.` (b) `Parallel to Parallel.` (c) `Parallel to Serial.`

Figure 5. Overhead of data movement between serial and parallel sections on our system (`IBA`) and TreadMarks (`TMK`).

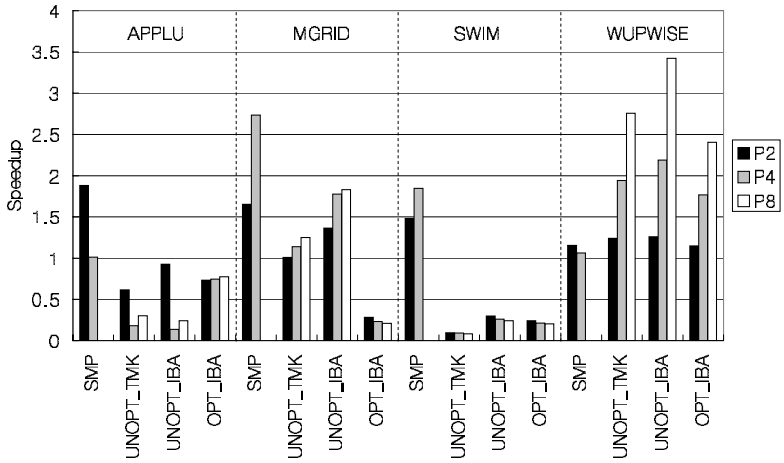


Figure 6. Speedup of the OpenMP applications.

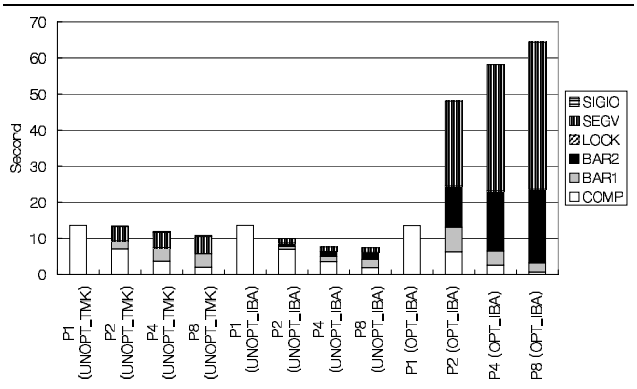


Figure 8. Overhead analysis in `MGRID`.

most overheads on the DVSM systems result from page locality, and the serialization of parallel regions largely degrades the performance. One of the solutions to alleviate this problem is to execute serial code sections also on child processes [10]

3.2.5. SWIM Figure 9 shows execution overhead in `SWIM`. Most loops are parallelized in an unoptimized version, but many page faults at the serial sections just after `calc1_do_1` and `calc2_do_1` results in significantly large overhead. The optimized code makes this problem worse.

3.2.6. WUPWISE Figure 10 shows overhead metrics of `WUPWISE` execution. There is very a few small parallel regions in this benchmark, and there is no noticeable difference in unoptimized and optimized versions. Most of `SEGV` overhead is generated at the serial section after the `zcopy_do_1` parallel region.

4. Conclusion

In this paper, we evaluated performance of SPEC OMP benchmarks on the next generation network technology based DVSM system, i.e. the InfiniBand Architecture. From experiment we showed that the next-generation of network technologies greatly improve the performance over the tra-

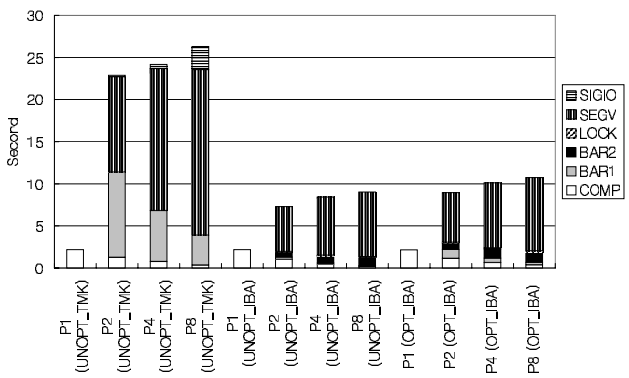


Figure 9. Overhead analysis in `SWIM`.

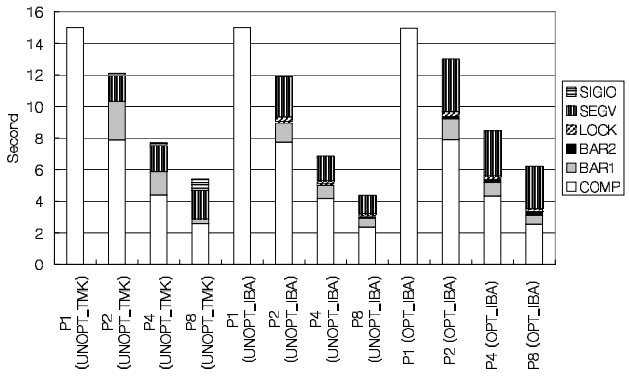


Figure 10. Overhead analysis in `WUPWISE`.

ditional implementation of DVSM systems. The performance gain results from using the remote DMA features, which does not involve any process interruption for data communication. Also we showed that the serialization of parallel code sections to reduce parallel code overhead results in severe overhead on the DVSM systems due to more data movement between serial and parallel sections. Even if the new network architecture helps to reduce the communication overhead, the application performance is very sensitive to the data locality. Since the standard OpenMP specification does not provide a mechanism to control data distribution, we need more advanced techniques to predict page movement and prefetch pages correctly inside the DVSM for better data locality and hiding latency.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] V. Aslot. Performance characterization of the SPEC OMP benchmarks. Master's thesis, Electrical and Computer Engineering, Purdue University, May 2002.

[3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. *Lecture Notes in Computer Science (WOMPEI2001)*, 2104:1–10, 2001.

[4] I. T. Association. *InfiniBand Architecture Specification, Release 1.0*. October 2000.

[5] A. Basumallik, S. J. Min, and R. Eigenmann. Towards OpenMP execution on software distributed shared memory systems. *Lecture Notes in Computer Science (WOMPEI'2002)*, 2327:457–468, 2002.

[6] T. Birk, L. Liss, and A. Schuster. Efficient exploitation of kernel access to InfiniBand: a software DSM example. In *Hot Interconnects*, Stanford, CA, August 2003.

[7] A. L. Cox, E. de Lara, C. Y. C. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols for software shared memory. In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, 1999.

[8] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarthy, W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory access networks. In *The 24th International Symposium on Computer Architecture (ISCA-24)*, June 1997.

[9] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and implementation of MPICH2 over infiniband with RDMA support. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[10] H. Lu, A. L. Cox, and W. Zwaenepoel. Contention elimination by replication of sequential sections in distributed shared memory programs. *ACM SIGPLAN Notices*, 36(7):53–61, 2001.

[11] S. J. Min, A. Basumallik, and R. Eigenmann. Supporting realistic OpenMP applications on a commodity cluster of workstations. *Lecture Notes in Computer Science (WOMPAT2003)*, 2716:170–179, 2003.

[12] S. J. Min, S. W. Kim, M. Voss, S. I. Lee, and R. Eigenmann. Portable compilers for OpenMP. *Lecture Notes in Computer Science*, 2104:11–19, 2001.

[13] OpenMP Forum, http://www.openmp.org/. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, October 1997.

[14] I. Park, S. W. Kim, and K. Park. Characterization of OpenMP applications on the InfiniBand-based distributed virtual shared memory system. In *Accepted at the International Conference on High Performance Computing*, 2004.

[15] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design, simulations, implementation and performance. In *The Fourth IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Jan. 1998.

[16] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for a SMP cluster. In *The 1st European Workshop on OpenMP (EWOMP'99)*, pages 32–39, September 1999.

pSigma: An Infrastructure for Parallel Application Performance Analysis using Symbolic Specifications

Simone Sbaraglia, Kattamuri Ekanadham, Saverio Crea
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{*sbaragli,eknath,screa*}*@us.ibm.com*

Seetharami Seelam
University of Texas at El Paso
El Paso, TX 79968, USA
seelam@cs.utep.edu

Abstract

In this paper we describe **pSigma***, which is an infrastructure for instrumenting parallel applications, enabling the users to probe into the execution of an application by intercepting its control-flow at selected points, the selection being made by lucid specifications using symbolic names of data structures and functions in the source program and performance-related information about the memory subsystem. This infrastructure assists application developers and system architects in examining performance events and relating them to the data and control components at the source program level. By providing details on how data is shared among parallel program components, it helps in exposing the performance effects induced by mutual interference or cooperation. Our event-based selection schema using symbolic names from the source domain results in a very user-friendly instrumentation mechanism.*

1 Introduction

A variety of performance measurement, analysis, and visualization tools [3, 4, 5, 6] have been created to assist application programmers in tuning and optimizing scientific codes. However, there is still very limited support for fine-grained performance analysis of parallel programs running on real machines. Understanding and tuning memory system performance of scientific applications running on parallel machines is still a critical issue for programmers and architects.

Existing performance analysis and visualization tools tend to be *control-centric*, since they focus on the control structure of the program (e.g. loops and functions). However, users also need *data-centric* performance measurement infrastructures that can help them understand the precise memory references that are causing poor utilization of memory hierarchy, especially in a parallel environment.

In order to develop such performance tools, tool developers and researchers often rely on *instrumentation libraries* [2, 3, 11, 12] to intercept the execution of a program at selected events and insert *probe libraries* for measurement and analysis. Unfortunately, such instrumentation libraries are not "performance-oriented", in that they do not provide the probe library with any information about the state of the machine, or the effect of the current operation on the system. For example, although they allow the probe library to be invoked upon the execution of a memory operation, they do not provide any information about the effect of the operation on the memory subsystem, such as whether it is a hit/miss in the cache, or whether it resulted in issuing any prefetches etc. Similarly, when the probe library is invoked as a result of a load operation, no symbolic information is provided as to which element and which data structure is being accessed.

Tool developers need flexible mechanisms to intercept the execution of a program at selected events based on information that relates performance to symbolic entities in the source program. This information must encompass execution events, such as accesses to array elements, as well as performance events such as cache misses exceeding a certain limit.

In this paper we present a new instrumentation infrastructure, which is an evolution of the **SiGMA** infrastructure for memory analysis, described in [1], to inject arbitrary user code into a parallel application binary. The user can describe the rules that should trigger the execution of the injected code using a specification that employs symbolic and performance related metrics. The infrastructure provides information at the granularity of individual load/store events for each thread of execution of an application and can be used for understanding sharing of data among parallel threads and the

resulting performance effects.

The remainder of this paper is organized as follows: In Section 2 we discuss the **pSigma** infrastructure for instrumentation of parallel applications. In Section 3 we describe the symbolic specification of events, in Section 4 we discuss the standardized event description interface to pass information to the event handler and in Section 5 we provide an illustrative example of use of **pSigma**. Finally, in Section 6, we describe one possible application of the infrastructure, to detect performance critical data structures in an OpenMP benchmark code. We present our conclusions in Section 7.

2 The pSigma Infrastructure

Figure 1 presents an overview of **pSigma** and the execution environment. On the top left corner of Figure 1, the three principal components of **pSigma** software are shown: a catalogue of *events*, a library of *event-handlers* and the module *psigmaInst*. An application user consults the event catalogue and composes a script describing a configuration for the memory subsystem, a list of events to be monitored, and the associated event-handlers, which can be picked from the **pSigma** library or supplied by the user. The *psigmaInst* module takes the script, the application binary and any user-supplied event-handlers and produces an instrumented binary that performs the desired probing.

Events can be of two types: Execution-specific events identify actions specified by the source program, such as the execution of a load, store or floating point instruction using a chosen data structure, or during the invocation of a certain function. Performance-specific events identify points during execution when certain criteria of performance metrics are satisfied, such as cache miss count exceeding a certain limit, an instruction or loop repeating a specified number of times.

The bottom of Figure 1 depicts the scenario when the instrumented binary is executed. The instrumented binary executes possibly multiple threads and whenever a designated event occurs, it generates an event-description packet. Memory related events are passed to the memory simulator, which simulates the cache structure for each processor. The packets carry the thread identification, so that corresponding caches are manipulated. After simulating the event in the cache structure, it generates event-descriptors that carry the results of cache accesses. All event-data packets are directed to the corresponding event-handlers that consume the data. All event-descriptors adhere to a standardized event-data format so that handlers can be developed using the uniform interface format. Event-data includes the identification of the data and control locations, in terms of symbolic names used for the data structures and functions in the source program. For instance, it uniquely identifies the subject data element and array name in the program and the line number and function of the source program that is manipulating it during that event. Memory simulators can also optionally provide cache subsystem state, giving access results (hit/miss) at each level, information on evicted elements on cache fills and list of prefetched data elements.

An event handler can use the event information in arbitrary ways. Handlers can accumulate data and produce summaries at the end or enumerate the data and produce traces and profiles. The **pSigma** library provides a host of event handlers, that typically summarize event-data: they can provide compressed traces of execution, enumerations of selected event data, summaries of cache behavior, categorized by functions and data structures, etc.

In addition, **pSigma** system also provides a facility to specify memory subsystem parameters and data structure paddings that can be simulated. The handlers can also dynamically turn on/off the generation/handling of certain events based on the occurrence of other events.

3 Symbolic Specification of Events

To facilitate usability and for fine grained instrumentation of selected parts of a program **pSigma** provides an easy scripting language for expressing desired actions. The specification is a set of *directives*, each specifying an event and a corresponding action to be taken when that event occurs. The event can be one event or a logical combination of several events. An event is a predicate on the current operation (*eg.* operation is a load or it suffered an L1 miss), or a predicate on the accumulated state of the system (*eg.* accumulated L1 Miss count exceeds a constant). The library supplies a list of events and counters that are tracked in **pSigma**. An event can also be further qualified by a *context* which can specify that the event must be considered only when it relates to a named data structure and/or occurs within the execution of a specified function. An action can invoke a handler or print a counter.

directive ::= **on** *compoundEvent* **do** *action*
compoundEvent ::= event | *compoundEvent* **and** *event*
| *compoundEvent* **or** *event*
event ::= currentEvent [*context*] | *counterState* **relOp** *value*
value ::= counterState | *constant*
currentEvent ::= load | *store* | *L1Miss* | *L2Hit* | ...
counterState ::= counterName [*context*]
counterName ::= cumL1Misses | *cumL2Hits* | ...
context ::= **for** *structureName* **in** *functionName*
action ::= **call** *handlerName* | **print** *counterName*

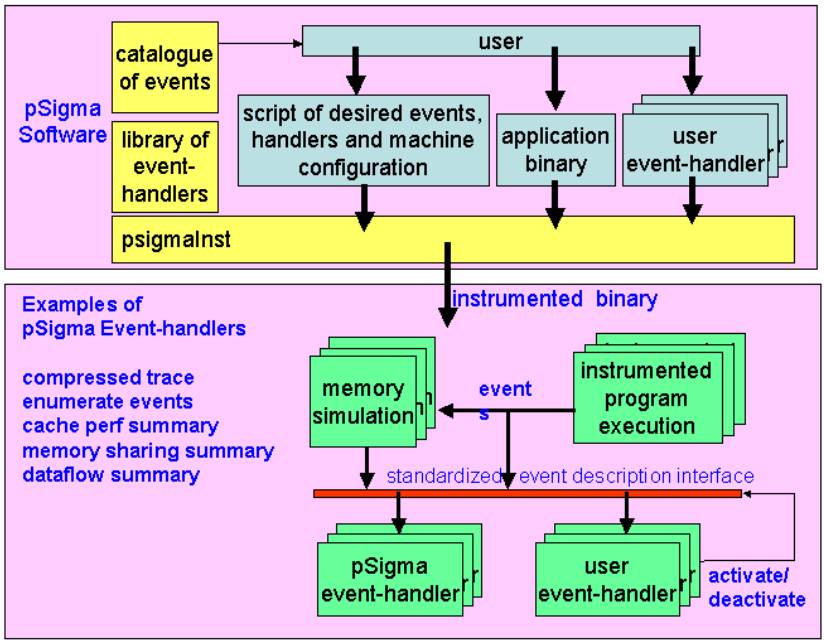


Figure 1. Structure of pSigma

Examples:

1. **on** *L1Miss* **for** *A* **in** *F* **do** *call myHandler*
2. **on** *L1Miss* **for** *A* **in** *F* **and**
   (cumL1Misses > 1000) **for** *A* **in** *F*
   do *call myHandler*
3. *pSigmaInst -p scriptFile myHandlers.a appbin*

In the first example, myHandler is invoked whenever an access to *A* in function *F* incurs a miss in L1. In the second example, the handler is called for each L1 miss on *A* in *F* after 1000 such misses have incurred. The third example illustrates how pSigma is invoked, supplying a script file, the library of user-supplied handlers and the application binary. The script file contains the event directives and the memory configuration of the system. The pSigmaInst command instruments the application binary as specified by the script and generates a new binary *app.inst* which includes all the necessary libraries. When this is executed, the handlers are invoked when the associated events occur.

Thus **pSigma** enables easy specification of very complex combination of conditions and provides sophisticated selection of events for which the user can obtain very detailed information on cache states and side-effects during execution. Using this infrastructure, users can quickly implement intelligent probes to get to the bottom of a performance problem.

4 Event Handler Interface

When the execution of an application is intercepted, **pSigma** provides the following information to the corresponding event handler. This interface is standardized so that all event handlers, including those to be supplied by the users, can be written for this interface. The event descriptor structure has three main components:

1. *InstructionData*: This includes the virtual address of the instruction that triggered the event, the opcode/type of the instruction (load, store, float etc.), the source line number and file to which this instruction belongs.

2. *MemoryData*: This indicates hit/miss at each level of cache and TLB that occurred when this instruction is executed, a list of evicted lines, prefetched lines that took place at each cache level during the execution of this instruction.

3. *CumulativeData*: This includes performance counters accumulated during program execution for each data structure and for each function.

5 An Illustrative Example

To illustrate the use of **pSigma**, we analyze here a simple Fortran sequential kernel, whose main loop is

```
parameter (nx=512, ny=512)
real a1(nx,ny), a2(nx,ny), a3(nx,ny)
do iter = 1, niters
  do id = 1, nd
    do jd = 1, nd
      do iy = 1, ny
        do ix = 1, nx
          ixy = ix + (iy-1)*nx
          a1(ix,iy) = a1(ix,iy)
 &          + c11(ixy,id,jd)*b1(ixy,jd)
 &          + c12(ixy,id,jd)*b2(ixy,jd)
 &          + c13(ixy,id,jd)*b3(ixy,jd)
          a2(ix,iy) = a2(ix,iy)
 &          + c21(ixy,id,jd)*b1(ixy,jd)
 &          + c22(ixy,id,jd)*b2(ixy,jd)
 &          + c23(ixy,id,jd)*b3(ixy,jd)
          a3(ix,iy) = a3(ix,iy)
 &          + c31(ixy,id,jd)*b1(ixy,jd)
 &          + c32(ixy,id,jd)*b2(ixy,jd)
 &          + c33(ixy,id,jd)*b3(ixy,jd)
        end do
      end do
    end do
  end do
end do
```

Figure 2. Fortran Kernel

```
on load   for a1 do call myHandler
on store  for a1 do call myHandler
on load   for a2 do call myHandler
on store  for a2 do call myHandler
on load   for a3 do call myHandler
on store  for a3 do call myHandler
nCaches 2
TLBEntries 256
TLBAssoc 2-way
TLBRepl LRU
L1Size 64Kb
L1Linesize 128b
L1Assoc 128-way
L1Repl LRU
L2Size 4Mb
L2LineSize 128b
L2Assoc 2-way
L2Repl LRU
```

Figure 3. Event/Architecture Specification

shown in Figure 2. We instrumented the application to intercept all load and store operations for data structures *a1*, *a2* and *a3*. The memory system parameters used in this example are same as in the IBM Power3 system. The event description file is shown in Figure 3. On each load and store operation on the selected data structures *myHandler* is invoked. It accesses the *MemoryData* structure provided by the standardized event description interface and computes derived metrics such as hit and miss ratios of each cache level and for the TLB, as shown in Figure 4.

It can be observed that the three data structures incur a large number of TLB misses. A close analysis of the code reveals that this is due to the data structure layout, that is such that *a1*, *a2* and *a3* map to the same TLB blocks. Since the TLB, in this experiment, is only 2-way set associative and the three data structures are accessed in an interleaved manner, this layout causes a lot of misses. Most of these misses could be avoided by restructuring the data structure layout. Figure 5 illustrates a restructuring of the program, in which the matrices *a1*, *a2* and *a3* are padded so that they don't map to the same TLB block. This new layout incurs significantly less TLB misses, as shown in Figure 6.

```
                    L1       L2       TLB
DATA: a1
  Load Hit Ratio    96.87%   99.99%   7.42%
  Store Hit Ratio   99.22%   16.67%   -

DATA: a2
  Load Hit Ratio    96.88%   50.00%   7.44%
  Store Hit Ratio   99.22%   16.67%   -

DATA: a3
  Load Hit Ratio    96.88%   59.99%   0.14%
  Store Hit Ratio   99.22%   16.67%   -
```

Figure 4. Memory Profile for a1,a2,a3

```
parameter (nx=512, ny=512)
real a1(nx+1,ny), a2(nx+1,ny), a3(nx+1,ny)
do iter = 1, niters
...
```

Figure 5. Restructured Fortran Kernel

```
                    L1       L2       TLB
DATA: a1
  Load Hit Ratio    96.87%   10.02%   99.90%
  Store Hit Ratio   99.22%   15.53%   -

DATA: a2
  Load Hit Ratio    96.87%   34.22%   99.63%
  Store Hit Ratio   99.22%   14.88%   -

DATA: a3
  Load Hit Ratio    96.87%   33.29%   96.51%
  Store Hit Ratio   99.22%   14.42%   -
```

Figure 6. Memory Profile for New a1,a2,a3

6 OpenMP programs on pSigma

When threads concurrently executing on multiple processors make accesses to shared data, there are many opportunities for performance analysis and tuning. There are several degrees of freedom that can be exploited to improve performance in this area: careful scheduling of work items to threads, placement of threads on different processors and placement of data in the address space. All these decisions interact with the data sharing pattern dictated by the application and

Array	Function	Memory References	Stores	Cold Misses	c2c transfers
p	conj_grad@OL@B@OL@F	741241600	0	2624	1048576
a	conj_grad@OL@B@OL@F	741241600	0	16588	70057
z	conj_grad@OL@B@OL@13	29649664	0	2621	39427
z	con_grad@OL@B@OL@11	11200000	5600000	0	2395
colidx	conj_grad@OL@B@OL@F	741241600	0	0	403

Table 1. cache to cache transfers in NAS CG

```
1 !$omp do
2   do j=1,naa+1
...
3     p(j) = r(j)
4   enddo
...
5 !$omp do
6  do j=1,lastrow-firstrow+1
7    sum = 0.d0
8    do k=rowstr(j),rowstr(j+1)-1
9      sum = sum + a(k)*p(colidx(k))
10   enddo
11   q(j) = sum
12 enddo
```

Figure 7. Segment of the code from CG that had a large number of cache line transfers

influence the performance. For instance, when a producer and consumer of a data item are scheduled to run on different processors, the corresponding cache line will have to be transferred between the two processors, which could be avoided if the producer and consumer are assigned to the same thread or to the same processor. Another example is that of false sharing [7, 9, 8], where two threads access disjoint sets of data from the same cache line, but do not necessarily share any data from it. Because of coherence policies, this can result in the cache line ping-ponging between two processors, which can be avoided under some conditions. Analysis of such situations for potential performance improvements can be complex. But it must be aided by tools that provide detailed accounts of what is happening. Our goal is to provide an infrastructure for these studies where **pSigma** will be able to expose potential opportunities and also provide simulated results when data is repartitioned, work is repartitioned among threads and schedules are rearranged.

In this section we provide an example of a simple observation of how many times cache lines are being transferred between processors. Some of these may be avoided and some may not be. Here we just provide a count for all such transfers. Our tool, known as *c2c*, provides a count of cache-to-cache transfers under the following conditions. It assumes an infinite cache system. The first time a cache accesses a line, that is counted as a cold access. All other accesses, ideally should not cause a fault, under the infinite cache assumption. However, in a multi-processor situation, a writer invalidates the line from all other caches and hence the caches may have to obtain the line again for further accesses. Since all caches are infinite, all these will be cache to cache transfers. This count gives an idea about the flow of information and can be used to guide the schedules and data mapping.

We instrumented the CG OpenMP benchmark from NPB ([10]) suite. A segment of this code, which is repeated a large number of times, is shown in Figure 7. Two loops manipulate the array *p*. The first loop initializes it and the second loop accesses it through indirection. Both loops are parallelized (statically scheduled) by OpenMP directives. Since the producer and consumer for the same data may run on two different processors, this will cause some cache to cache transfers. Table 1 shows the transfer counts categorized by function and data structure. Our tool indicated that the data structure p has suffered more than 80% of the total invalidations in the program. This information is useful for tuning the application for improved performance.

7 Conclusion

In this paper we presented the **pSigma** infrastructure for instrumentation of parallel applications. This infrastructure allows the user to probe into the execution of the application at selected points, which can be specified in a natural language using symbolic names of data structures and functions and performance-related information. Furthermore, the infrastructure transparently provides performance-related metrics which can assist in developing data-centric performance monitoring and analysis tools.

We demonstrated one possible application of the

infrastructure by implementing a tool which detects cache-to-cache transfers in OpenMP applications.

References

[1] L. Derose, K. Ekanadham, J.K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of Supercomputing '02*, November 2002.

[2] Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

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

[4] Luiz DeRose and Daniel Reed. Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.

[5] Bernd Mohr, Allen Malony, and Janice Cuny. TAU Tuning and Analysis Utilities for Portable Parallel Programming. In G. Wilson, editor, *Parallel Programming using C++*. M.I.T. Press, 1996.

[6] Jerry Yan, Sekhar Sarukkai, and Pankaj Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software Practice & Experience*, 25(4):429-461, April 1995.

[7] Tor E. Jeremiassen , Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *ACM SIGPLAN Notices*, 30(8):179-188, August 1995.

[8] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, Per Stenstrm, The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th annual international symposium on Computer architecture*, p.88-97, May 16-19, 1993, San Diego, California, United States.

[9] Joseph Torrellas , Monica S. Lam , John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches, *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[10] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. *NAS Technical Report NAS-99-011*, NASA Ames Research Center, Moffett Field, CA, 1999.

[11] S.J. Eggers, D.R. Keppel, E.J. Koldinger, and H.M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS conference on On Tue, 28 Sep 2004, Simone Sbaraglia wrote: Measurement and Modeling of Computer Systems*, p.37-47, April 1990.

[12] Jordi Caubet, Judit Gimenez Jesus Labarta, Luiz DeRose, and Jeffrey Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools- WOMPAT 2001*, pages 53 – 67, July 2001.

Session 2: Application studies

Chair: Sanjiv Shah, Intel

- M. Süß and C. Leopold, A User's Experience with Parallel Sorting and OpenMP
- C. Jaillet and M. Krajecki, Constructing Optimal Golomb Rulers in Parallel

A User's Experience with Parallel Sorting and OpenMP

Michael Süß and Claudia Leopold
University of Kassel
Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
{msuess, leopold}@uni-kassel.de

Abstract

Some algorithmic patterns are difficult to express in OpenMP. In this paper, we use a simple sorting algorithm to illustrate problems with recursion and the avoidance of busy waiting. We compare several solution approaches with respect to programming expense and performance: stacks, nesting and a workqueue (for recursion), as well as condition variables and the sched_yield–function (for busy waiting). Enhancements of the OpenMP–specification are suggested where we saw the need. Performance measurements are included to backup our claims.

1. Introduction

Parallel Programming is still a challenging task in our days. Although there are many powerful parallel programming systems, most of them operate on a relatively low abstraction level (e.g. *POSIX threads*, *MPI*). The specification of OpenMP [2] promised advances in this regard, and has provided a relatively smooth way to incrementally parallelize existing programs as well as to write powerful new applications on a high abstraction level, since its introduction in 1997. Its portability and vendor acceptance quickly helped the system to become a de facto standard for programming shared memory parallel machines.

Nevertheless, OpenMP is not without problems and rough edges, which version 2.0 of the specification was not able to straighten out fully. This paper uses a simple sorting algorithm to show some of these, as well as a couple of techniques developed to work around them. All examples were written in the *C++* programming language. Some suggestions for future enhancements to the specification are included.

The next section gives a short summary of the used sorting algorithm. Section 3 describes the two problematic areas that we encountered during the course of our work: *recursion* and *busy waiting*. Figure 1 depicts a diagram that

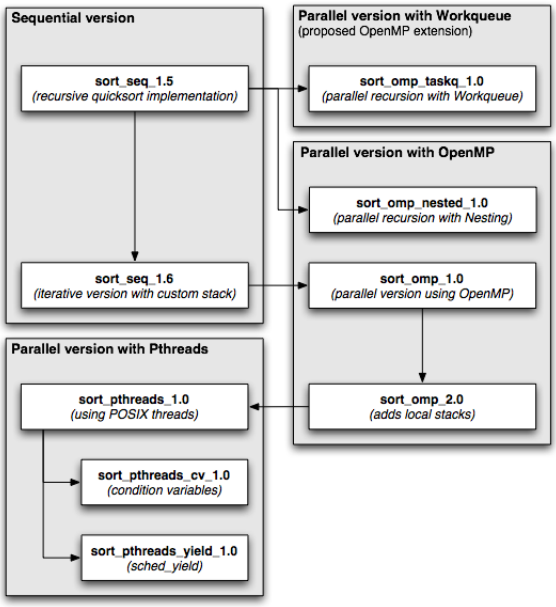


Figure 1. Revision tree of Sorting Program

illustrates the history of the different program versions we wrote to solve these problems (some less important versions are left out in the diagram, causing missing entries in the numbering scheme). Three different solution strategies are presented for recursion:

- an iterative approach (*sort_omp_1.0*)
- an advanced iterative version using local stacks (*sort_omp_2.0*)
- nested parallelism (*sort_omp_nested_1.0*)
- a *workqueue* (*sort_omp_taskq_1.0*)

The problem of busy waiting is approached with solutions borrowed from *POSIX threads*: *condition variables* (*sort_pthreads_cv_1.0*) and the *sched_yield*–function

(*sort_pthreads_yield_1.0*). Ideas for extensions to the OpenMP–specification presented in section 3 are the addition of an *omp_get_num_all_threads*–function, condition variables and an *omp_yield*–function. Note that at this point only data points are given for the inclusion of these extensions into the specification, as this paper merely describes our experiences with a certain problem area and does not include a complete proposal. Section 4 discusses the performance of our solutions. In section 5, we sum up our results and discuss further perspectives.

2. Sorting

Sorting data has always been one of the key problems of computer science. Many sequential algorithms have been suggested, of which *Quicksort*, invented and described by Hoare [4], is one of the most popular ones. It works recursively using the *divide and conquer* principle:

1. choose a pivot element, usually by just picking the last element out of the sorting area
2. iterate through the elements to be sorted, moving numbers smaller than the pivot to a position on its left, and numbers larger than the pivot to a position on its right, by swapping elements. After that the sorting area is divided into two subareas: the left one contains all numbers smaller than the pivot element and the right one contains all numbers larger than the pivot element.
3. goto step *1* for the left and right subareas (if there is more than one element left in that area)

The algorithm has an average time complexity of $O(n \cdot log(n))$ and a worst case time complexity of $O(n^2)$, making it one of the fastest sequential sorting algorithms available. Since we do not aim at producing the fastest parallel sorting algorithm ever, but merely try to show some problems and solutions with OpenMP, we chose this easy and widely used algorithm as the basis for our experiments, instead of a more advanced and complex (sequential or parallel) one. Previous experiments (although with a different focus) with quicksort and OpenMP have been conducted by Parikh [6].

3. A simple sorting algorithm and its problems

The base version for our tests was the simple quicksort algorithm described above, combined with *insertion sort* for small sorting areas. This algorithm (*sort_seq_1.5*) is sketched in Figure 2. We tried to speed up the algorithm through advanced (sequential) sorting techniques, but the performance gain was not worth the loss in simplicity.

Using this algorithm, we started our experiments with OpenMP, and soon after were confronted with our first problem: *recursion*.

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r)
{
    T pivot;
    int i, j;

    /* only small segment of numbers left -> use insertion sort! */
    if (r - q < SWITCH_THRESH) {
        myInsertSort(myVec, q, r);
        return;
    }

    /* choose pivot, initialize borders */
    pivot = myVec[r];
    i = q - 1;
    j = r;

    /* partition step, which moves smaller numbers to the left
       and larger numbers to the right of the pivot */
    while (true) {
        while (myVec[++i] < pivot);
        while (myVec[--j] > pivot);
        if (i >= j) break;
        std::swap(myVec[i], myVec[j]);
    }
    std::swap(myVec[i], myVec[r]);

    /* recursively call yourself with new subsegments,
       i is index of pivot */
    myQuickSort(myVec, q, i - 1);
    myQuickSort(myVec, i + 1, r);
}
```

Figure 2. Base version of the quicksort algorithm

3.1. Problem 1: Recursion

There is no easy and intuitive way to deal with recursion in OpenMP (yet), as the basic worksharing constructs provided by the specification (*for* and *sections*) do not seem to be well suited for recursive function calls. Our first solution to the problem involved getting rid of the recursion, as already suggested by an Mey [1]. It is a widely known fact that, by using a stack, every recursion can be changed into an iteration, and that is exactly what we did in *sort_seq_1.6*. This step was one of the most time consuming of all, as it involved the introduction of a new data structure, specially tailored to our problem. We called this structure *globalTodoStack*, it stores the intervals still to be sorted.

This program was later parallelized into *sort_omp_1.0*, making the following changes:

- a *parallel* region was added in *main*, around the first call of the *myQuickSort*–function
- a call to *omp_get_thread_num* was added, so that only one thread initially executes the *myQuickSort*–function, all others wait until work for them is put on the *globalTodoStack*
- all accesses to shared variables (especially the *globalTodoStack*) were protected by *critical* sections to

prevent multiple threads from accessing the variables at the same time

The resulting program is sketched in Figure 3.

In *sort_omp_2.0* the global stack was complemented with one local stack per thread. All new segments to be sorted are put on the local stacks per default. Only when one of the local stacks is empty, it needs to communicate with the global stack and poll for new work there. It works the other way around as well: when the global stack is getting empty, new work is pushed on it from a local stack. This modification led to a significant performance gain, because the need for synchronisation via critical sections dropped considerably. The effect is most visible for high numbers of working threads, because that is when a lot of work sharing needs to be done.

Our second solution to the recursion problem (*sort_omp_nested_1.0*) involves nested parallelism as illustrated in Figure 4. There are a few problems with this approach though. First of all, the OpenMP–specification allows compilers to serialize nested parallel directives, and many still do so. With these compilers, the version will not achieve considerable speedups, performance portability is not granted. Furthermore, nesting support in the specification seems to be a little immature. In particular, the *omp_get_num_threads*–function is useless for our example, as it always returns two threads per parallel region. The value we were really interested in is the number of all running threads (to limit the creation of new threads as we dive deeper into the recursion). Since this number is not available through a simple function call, we needed to track it ourselves, therefore introducing new (and performance hindering) critical sections into the code. Perhaps this could be taken care of by the introduction of an *omp_get_num_all_threads* function call or some similar mechanism into the next iteration of the specification.

The third solution to the recursion problem has been first suggested by Shah et al. [7]. It involves usage of the *workqueuing model* (*sort_omp_taskq_1.0*, depicted in Figure 5). This model has been proposed as an OpenMP–extension, but has not been accepted yet. To the authors knowledge, only two compilers understand the new pragmas, and so this solution has the drawback that the code presented here is not portable. This might change quickly though, if the *workqueuing*–proposal is accepted. Except for this drawback, the solution is easy and elegant. This version lacks the ability to add *local queues* (similar to the local stacks that brought a performance improvement in *sort_omp_2.0*), but a smart compiler might recognize this performance potential and insert them automatically.

Having solved the recursion problem, the next one appeared: *busy waiting*.

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r,
      std::stack <std::pair <int, int>>&globalTodoStack,
      int &numBusyThreads, const int numThreads)
{
     bool idle = true;

     /* Skipped: Initialisation */

     while (true) {

          /* only small segment of numbers left ->use insertion sort! */
          if (r - q < SWITCH_THRESH) {
               myInsertSort(myVec, q, r);
               /* and mark the region as sorted, by setting q to r */
               q = r;
          }

          while (q >= r) {   /* Thread needs new work */

               /* only one thread at a time should access the
                  globalTodoStack, numBusyThreads and idle variables */
               #pragma omp critical
               {
                    /* something left on the global stack to do? */
                    if (false == globalTodoStack.empty()) {
                         if (true == idle) ++numBusyThreads;
                         idle = false;
                         /* Skipped: Pop a new segment off the stack */
                    } else {
                         if (false == idle) --numBusyThreads;
                         idle = true;
                    }
               }

               /* if all threads are done, break out of this function.
                  note, that the value of numBusyThreads is current, as there
                  is a flush implied at the end of the last critical section */
               if (numBusyThreads == 0) {
                    return;
               }
          }

          /* Skipped: choose pivot and do partitioning step */

          #pragma omp critical
          {
               globalTodoStack.push(pair(q, i - 1));
          }

          /* iteratively sort elements right of pivot */
          q = i + 1;
     }
}

int main(int argc, char *argv[])
{
     /* Skipped: Program Initialisation */

     #pragma omp parallel shared(myVec, \
      globalTodoStack, numThreads, numBusyThreads)
     {
          /* start sorting with one thread, the others wait for the stack to fill up */
          if (0 == omp_get_thread_num()) {
               myQuickSort(myVec, 0, myVec.size() - 1,
                 globalTodoStack, numBusyThreads,
                 numThreads);
          } else {
               myQuickSort(myVec, 0, 0,
                 globalTodoStack, numBusyThreads,
                 numThreads);
          }
     }
     /* Skipped: Tests and Program output */
}
```

Figure 3. Parallel part of sort_omp_1.0

```
template < typename T >
void myQuickSort(std::vector < T > &myVec, int q,
        int r, int &numBusyThreads, const int numThreads)
{
    /* Skipped: Initialisation + Partitioning step */

    /* do not nest, if there are too many threads already */
    if (numBusyThreads >= numThreads) {
        myQuickSort(myVec, q, i - 1, numBusyThreads,
          numThreads);
        myQuickSort(myVec, i + 1, r, numBusyThreads,
          numThreads);
    } else {
        #pragma omp atomic
        numBusyThreads += 2;

        #pragma omp parallel shared(myVec, numThreads, \
         numBusyThreads, q, i, r)
        {
            #pragma omp sections nowait
            {
                #pragma omp section
                {
                    myQuickSort(myVec, q, i - 1,
                      numBusyThreads, numThreads);
                    #pragma omp atomic
                    numBusyThreads--;
                }
                #pragma omp section
                {
                    myQuickSort(myVec, i + 1, r,
                      numBusyThreads, numThreads);
                    #pragma omp atomic
                    numBusyThreads--;
                }
            }
        }
    }
}
```

Figure 4. Parallel part of sort_omp_nested_1.0

```
template < typename T >
void myQuickSort(std::vector <T> &myVec, int q, int r)
{
    /* Skipped: Initialisation + Partitioning step */

    #pragma omp taskq
    {
        #pragma omp task
        {
            myQuickSort(myVec, q, i - 1);
        }
        #pragma omp task
        {
            myQuickSort(myVec, i + 1, r);
        }
    }
}

int main(int argc, char *argv[])
{
    /* Skipped: Program Initialisation */

    #pragma omp parallel shared (myVec)
    {
        #pragma omp taskq
        {
            #pragma omp task
            {
                myQuickSort(myVec, 0, myVec.size() - 1);
            }
        }
    }

    /* Skipped: Tests and Program output */
}
```

Figure 5. Parallel part of sort_omp_taskq_1.0

3.2. Problem 2: Busy Waiting

When a thread has nothing to do in our sorting application (because both its local stack and the global stack are presently empty, e.g. at the beginning), it does not mean that it is allowed to quit. New tasks can be put on the global stack at any time and should of course be processed as soon as possible. The only way this can be accomplished with OpenMP is *busy waiting*, which means that the thread is constantly polling for work, wasting processor cycles that could be better spent in another thread.

For possible solutions, one must only look as far as to the *POSIX threads* standard. A synchronisation primitive called *condition variable* is implemented there and solves the problem in a quite hard to understand way (when one is looking at it through the eyes of a beginner to parallel programming), but nevertheless fully. Butenhof describes condition variables like this:

> *A condition variable is a "signaling mechanism" associated with a mutex and by extension is also associated with the shared data protected by the mutex.* Waiting *on a condition variable atomically releases the associated mutex and waits until another thread* signals *(to wake up one waiter) or* broadcasts *(to wake all waiters) the condition variable. The mutex must always be locked when you wait on a condition variable and, when a thread wakes up from a condition variable wait, it always resumes with the mutex locked.*
> ([3, p. 72])

The introduction of this concept into the OpenMP–specification has already been suggested by Lu et al. [5], and we would also like to encourage the inclusion of this or a similar mechanism into the specification. To illustrate the savings possible when using condition variables, we ported our sorting application to *POSIX threads* (staying as close to the original version as possible). The resulting program is *sort_pthreads_1.0*, which should perform about equal to *sort_omp_2.0* (see section 4 for details). Then, *sort_pthreads_1.0* was enhanced with condition variables (*sort_pthreads_cv_1.0*). Every time a thread finishes his work and finds nothing else to do on the stacks, it puts itself to sleep and is woken up by another thread only when there is new work to be done. This may lead to a significant performance gain on a heavily loaded machine.

A second and somewhat easier solution to the problem of busy waiting can be observed in *POSIX.1b* (realtime extensions). This standard defines a function *sched_yield*, which puts the calling thread at the end of the *ready–queue* of the operating system scheduler and selects a new thread to run. If there is no other thread waiting, the function returns immediately. The same could be done with a new function *omp_yield* in OpenMP. Whenever a thread runs out of work in our example program and the stacks are empty, it calls the suggested function. If other threads are waiting to be processed (which might not be out of work yet), these get a chance to run and produce more work for all idle threads. Though less powerfull than condition variables, our experiments with *sort_pthreads_yield_1.0* (a version incorporating *sched_yield*) suggest that this function is able to reduce busy waiting under heavy load for our problem as well (see Table 3).

4. Performance results

Performance tests were carried out on an otherwise unloaded node of an *AMD Opteron 848* class computer with 4 processors at 2.2 GHz, located at the RWTH Aachen. Programs were compiled with the *Intel C++ Compiler 8.1* using the *-O3 -openmp* compiler options. Further experiments were carried out on an otherwise unloaded node of a *Sun Fire 6800* class computer with a maximum of 8 *Ultra Sparc III* processors at 900MHz, also located at the RWTH Aachen. Here, the *Guide Compiler* by *Kuck & Assoc. Inc. (KAI)* was used with options: *-fast –backend -xchip=ultra3cu - -backend -xcache=64/32/4:8192/512/2 - -backend -xarch=v8plusb*.

Tables 1 and 2 show wall–clock time in seconds for all versions of our sorting program, with different numbers of threads. Only the time needed to actually perform the sorting algorithms was measured. All experiments were repeated at least three times, each time sorting 100 million random integers. For Tables 1 and 2, the best time achieved in each test was chosen.

Table 1 shows good speedups for all parallel program versions. The best performing solutions are *sort_omp_2.0* and the programs using Pthreads. Program *sort_omp_2.0* outperforms *sort_omp_1.0*, which shows the relevance of the local stacks. The programs with nesting and the workqueue are slower than the iterative programs, but this might be due to the relative immaturity of both options in the specification. The results for *sort_omp_nested_1.0* are to be taken with a grain of salt, since we were not able to fully control the number of threads used. Nevertheless, the best result we were able to achieve when running with different numbers of threads was 11.7 seconds, which is still slower than the other programs. No difference in speed is noticeable between the different versions using Pthreads. This is to be expected, as the advantages of these solutions will only show on heavily loaded systems or when looking at the CPU–time.

Program	Wall–clock time (sec.)		
	1Th.	2Th.	4Th
sort_seq_1.5	23.8	23.8	23.8
sort_seq_1.6	23.6	23.6	23.6
sort_omp_1.0	24.0	13.7	8.1
sort_omp_2.0	24.3	12.6	7.5
sort_omp_nested_1.0	23.9	21.4	12.4
sort_omp_taskq_1.0	29.8	16.3	9.1
sort_pthreads_1.0	24.0	12.7	7.5
sort_pthreads_cv_1.0	24.8	12.9	7.6
sort_pthreads_yield_1.0	24.5	12.7	7.5

Table 1. Wall–clock time for sorting 100 million integers on an AMD Opteron 2200 in seconds

Program	Wall–clock time (sec.)			
	1Th.	2Th.	4Th.	8Th.
sort_seq_1.5	36.8	36.8	36.8	36.8
sort_seq_1.6	37.4	37.4	37.4	37.4
sort_omp_1.0	38.2	23.9	15.7	11.0
sort_omp_2.0	37.9	21.4	13.1	10.0
sort_omp_nested_1.0	43.4	25.2	25.2	25.2
sort_omp_taskq_1.0	n.A.	n.A.	n.A.	n.A.
sort_pthreads_1.0	37.6	21.2	13.3	10.1
sort_pthreads_cv_1.0	37.2	20.3	13.3	9.6
sort_pthreads_yield_1.0	37.2	22.1	14.2	10.9

Table 2. Wall–clock time for sorting 100 million integers on a Sun Fire 6800 in seconds

The results in Table 2 look similar, except for two differences:

- nested parallel regions are serialized by the *Guide Compiler*, therefore no speedup beyond 2 is possible for *sort_omp_nested_1.0*
- we were not able to get *sort_omp_taskq* to work reliably on this platform (we have no idea if this is a compiler problem or a subtle bug in our implementation, but as soon as more than one thread was employed, it would either crash or run forever), therefore no results for this platform are provided

Table 3 demonstrates what happens on a heavily loaded system with and without *busy waiting*. The heavy load was built up by using four times as many threads as there are processors available on the machine. It shows average wall–

Program	SUN / 96Th.	AMD / 16Th.
sort_omp_1.0	> 600	20.5
sort_omp_2.0	> 600	19.0
sort_pthreads_1.0	15.9	7.8
sort_pthreads_cv_1.0	10.6	7.7
sort_pthreads_yield_1.0	11.4	7.9

Table 3. Average wall–clock time for sorting 100 million integers on heavily loaded systems in seconds

clock time to reduce the chance of lucky scheduling decisions.

On the SUN platform, the Pthreads solutions show the results we expected: Program *sort_pthreads_1.0* takes considerably longer than *sort_pthreads_cv* and *sort_pthreads_yield*. There are two relatively big surprises for us though: First, the OpenMP–versions are slow as compared to the *Pthreads*–versions (in case of the SUN machine so slow that we decided to cancel the runs after 10 minutes). The reasons for this are not yet clear to us and still under investigation. Second, on the *AMD*–machine, no performance difference is noticeable between the different *Pthreads*–versions. A better scheduler might account for this, but we are still investigating this question as well.

5. Concluding remarks and perspectives

In this paper, we have used several versions of a simple parallel sorting program to show some weaknesses of the OpenMP–specification and a couple of ways to address them. Suggestions for enhancements to the specification were made whenever it seemed appropriate.

Section 3.1 addressed the problem of *recursion*. Three solutions to it were presented, only one of which could be portably implemented with the present state of the specification, while the other, more elegant but less performant, two required additional support for the *workqueue* extension, or for nested parallelism, respectively.

In section 3.2, we have discussed the problem of wasted processor cycles (*busy waiting*). As possible solutions we suggest the introduction of *condition variables* and a new function *omp_yield*. Performance measurements showed that both approaches may provide adequate savings in processor time.

In the future, we plan to implement and test some of the ideas and additions to the specification we have suggested into an actual compiler, as well as investigate other algorithmic problems beyond sorting.

6. Acknowledgments

We are grateful to Björn Knafla and Beliz Senyüz for proofreading the paper. We thank the *Center for Computing and Communication* (especially Dieter an Mey) at the RWTH Aachen and the *University Computing Center* at the Technical University of Darmstadt for providing the computing facilities used to carry out our performance measurements.

References

[1] D. an Mey. Two OpenMP programming patterns. In *Proceedings of the Fifth European Workshop on OpenMP - EWOMP'03*, September 2003.

[2] O. A. R. Board. OpenMP specifications. `http://www.openmp.org/specs`.

[3] D. R. Butenhof. *Programming with POSIX Threads*. Addison–Wesley, 1997.

[4] C. Hoare. Quicksort. *The Computer Journal*, 5:10–15, 1962.

[5] H. Lu, C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Proc. of Supercomputing'98*, 1998.

[6] R. Parikh. Accelerating quicksort on the intel pentium 4 processor with hyper–threading technology. www.intel.com/cd/ids/developer/asmo-na/eng/technologies/threading/hyperthreading/20372.htm, 2003.

[7] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. In *Proceedings of the Fourth European Workshop on OpenMP - EWOMP'02*, September 2002.

Constructing Optimal Golomb Rulers in Parallel

Christophe Jaillet
Université de Reims Champagne-Ardenne
Département de Mathématiques et Informatique
Moulin de la Housse - BP 1039
F-51687 Reims Cedex 2
Email : christophe.jaillet@univ-reims.fr

Michaël Krajecki
Université de Reims Champagne-Ardenne
Département de Mathématiques et Informatique
Moulin de la Housse - BP 1039
F-51687 Reims Cedex 2
Email : michael.krajecki@univ-reims.fr

***Abstract*— In this paper, a construction of the Golomb optimal rulers is studied with a tree search approach. Improvements to the basic algorithm are given and it is parallelized using a shared memory. The application associated to this approach is written in C using the standard OpenMP and MPI libraries.**

The algorithm takes advantage of a collaboration mechanism between the processors. Different load balancing strategies are studied.

The application has proved efficient up to 32 processors. This solution opens up some new perspectives such as solving the already resolved instances of the problem more quickly, and solving next open instances in the future.

***Index Terms*— Golomb ruler, parallel algorithm, load balancing, shared memory, OpenMP, distributed memory, MPI**

I. Introduction

Golomb rulers are numerical sequences. The Optimal Golomb Rulers (*OGR*) construction is a hard combinatorial problem that consumes a high computation time. It represents a real *challenging* problem especially for parallel combinatorial search.

Although parallelization seems to be a good candidate to obtain further practical improvements, the research in this direction is not very developed. In a previous work, we have studied parallel resolution of CSP (Constraint Satisfaction Problems) with a shared memory [1]. The conclusions of that work provided a general approach to solve combinatorial problems in parallel and conducted the first choices made to construct optimal Golomb rulers.

The paper is organized as follows: the first section is dedicated to the Golomb rulers and introduces a classical approach.

The next section proposes a parallel shared memory solution for the construction. Then an implementation using OpenMP is introduced. The use of the schedule clause of the parallel *for* loop and of the OpenMP parallel region is discussed in detail.

Section IV gives experimental results. It illustrates the tasks allocation and the effect of collaboration between the processors. The results of the different load balancing strategies are compared and the impact of granularity is studied.

The paper ends with some conclusions on this work and some perspectives are introduced.

II. Golomb rulers

A Golomb ruler is an ordered sequence of integers, such that all differences between any two of them are different: considering that these *marks* refer positions on a linear scale, the *distances* on the *ruler* have all to be different [2] (see figure 1).

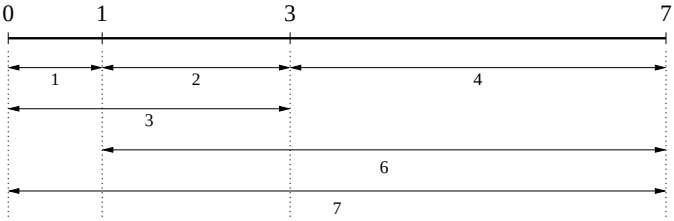


Fig. 1. A *non optimal* 4 marks Golomb ruler

It is easy to construct a ruler for any number n of marks, but the interesting cases deal with minimal length sequences [3] (*ig* (0,1,4,6) is optimal for 4 marks).

These sequences are a mathematical curiosity, but have also many applications in a wide variety of fields, including x-ray crystallography [4], self-orthogonal codes (for error detection and correction in coding theory) and communication (radio frequences, radio astronomy and PPM[1]).

[1]pulse phase modulation

This construction problem is a combinatorial one, and has been approached in different ways [5], [6].

For the rest of this document, let $G(n)$ represent the minimal length of an n marks Golomb ruler. Our goal is to compute that value (and give the constructed ruler). Once a ruler of a given length is found, the whole effort consists in finding a solution with a better length. This is the reason why we formulate the problem as "finding the optimal length under a given limit for the Golomb ruler".

A. A tree search algorithm

The construction of optimal Golomb rulers can be modelized as a tree search problem, with the view to minimize the length of the constructed sequence:

- as the length of an optimal n marks ruler does not exceed $1 + 2 + 4 + \cdots + 2^{n-2} = 2^{n-1} - 1$, it is possible to consider that the root value is 0 and that the values of the nodes are between 1 and $2^{n-1} - 2$ (see figure 2);
- every leaf of the tree symbolizes a sequence which is a solution if the values are ordered and if all the differences are different.

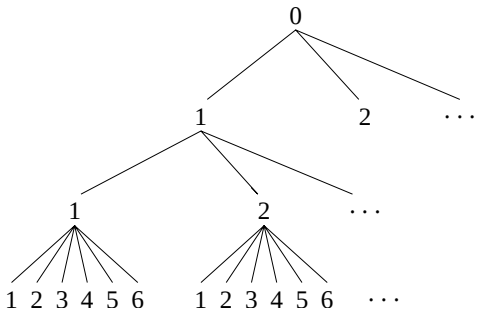


Fig. 2. Search tree for 4 marks Golomb rulers

It is now sufficient to propose a walk through the search tree, in depth first, keeping the best observed length, to get a simple sequential algorithm finding an optimal Golomb ruler.

This constructive algorithm belongs to the backtrack algorithms class; to be efficient, any recursive tree traversal must be avoided and the use of arrays as elementary data structure is strongly recommended.

B. Improvements

The search space can be reduced by applying some simple construction remarks (see figure 3):

- the best possible length $best\_length$ is set to $initial\_limit = 2^n - 1$ at the beginning (or it can be fixed at the execution time);
- when placing the k-th mark at position $pos(k)$, there are still $r = n - k$ remaining marks to be placed, which can't have a length $remaining\_length(r)$ less than $1 + 2 + \cdots + r = r(r+1)/2$ or than $G(r+1)$: it induces the following new constraint:

$$pos(k) + max(\frac{r(r+1)}{2}, G(r+1)) < best\_length$$

- mirror solutions can be avoided and thus the search tree reduced, by considering only sequences whose last distance is greater than the first one: if the second mark (first except 0) has already been set with the a value, and in order for the last distance b to satisfy $b \geq a + 1$, another constraint has to be introduced:

$$pos(k)+max(\frac{(r-1)r}{2}, G(r))+a+1 < best\_length$$

- as the best observed length decreases, more and more inconsistent values can be ignored.

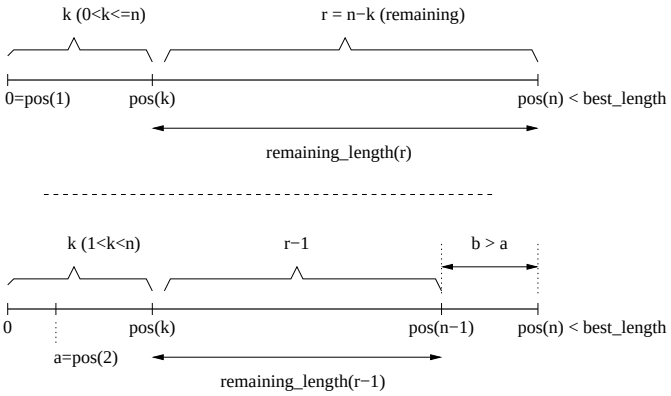


Fig. 3. New constraints based on construction remarks

According to the above global constraint, an efficient filtering algorithm is defined which eliminates remaining values in the the position variables domains in order to cut branches of the search tree. Note that it becomes a *Forward Checking* algorithm (*FC* in the field of CSP).

III. Golomb rulers parallel algorithm

In [7], we proposed to formalize the Langford problem as a CSP (*Constraint Satisfaction Problem*) and showed that an efficient parallel resolution is possible. In this section, we propose to develop a specific algorithm taking into account the conclusions of our previous works (management of the memory and load balance).

A. Tasks generation

The tree traversal induced by the explicit construction of all solutions can be made in parallel while introducing the following definition for the notion of task: it is associated to the traverse of a particular subtree. While choosing to develop all subtrees to a depth k, at most $(initial\_limit - 1)^k$ tasks can be defined.

Assume that this construction only keeps the *possible* tasks (those whose values are well ordered and satisfy the *all-diff differences* constraint, and which respect the search reduction constraints) that are accessible using a unique identifier.

The algorithm can be summarized in c-like mode by:

```
nbTasks = generateTasks(n,max,k);
best_length = initial_limit;
for(task=0 ; task<nbTasks ; task++)
  if ( useful(task,best_length) )
    best_length =
        solveTask(task,best_length);
```

where `nbTasks` is the number of initially useful tasks, deduced by the development of all subtrees to the depth `k` by the function `generateTasks` for `n` marks with `initial_limit` as initial best length. The function `solveTask` is in charge of traversing the subtree associated with the task numbered `task`. At the end, the variable `best_length` contains the optimal Golomb ruler length for the given value of `n`.

It can be noticed that, when introducing a backtracking scheme on the inconsistent branches (as soon as a first constraint is violated) the computations associated to these remaining tasks are particulary irregular.

B. Collaboration

The tasks may take advantage of the results of the previous ones by upgrading their value of $best\_length$, if such a collaboration scheme is used, because of the search space reduction.

When running a program in parallel, the tasks are balanced over the processors and each processor commonly has several tasks to compute. So there are three different levels of information: *global information*, *processor information*, and *task information*.

The goal of our application is to compute, for a given number of marks, the (global) minimum length of a Golomb ruler. It can be reached in three different manners:

- by keeping the best length value private to each task: when a processor gets a new task, it restores its previous value (but keeps the best of them for the final collaboration);
- by sharing the best length between all the tasks of each processor (the processors gather their last values before the program concludes): this enables to cut branches in the search tree, and even to directly evict some of the tasks (even in a sequental execution);
- by sharing the best length between all the tasks and updating it after each task: this solution provides an efficient way to decrease the best length used by each task; it may reduce the search space more quickly, but may impose an exchange overhead.

Our first experiments with OpenMP concerning the collaboration immediately proved that collaboration has to be used at least between the tasks, and that hardly any exchange overhead is induced by the full collaboration: the algorithm takes a real advantage of collecting the $Global\_best$ value as a shared information.

C. Parallel execution with OpenMP

The OpenMP environment has evolved to a standard for shared memory parallelism [8], [9]. It is a complete API for programming shared memory multiprocessors systems and makes it possible to obtain a parallel code easily (almost close to the sequential one).

1) Tasks allocation in a parallel loop: We will study the three following schedules: the *Static* repartition, the *Modulo Nbproc* static repartition, and the *Dynamic* one.

Thanks to OpenMP the parallel code is easy to define, adding a compilation directive before the choosen *for* loop instruction (here for a dynamic schedule with a full collaboration):

```
nbTasks = generateTasks(n,max,k);
Global_best = initial_limit;
#pragma omp parallel for
                  schedule(dynamic)
                  shared(Global_best)
for(task=0 ; task<nbTasks ; task++)
  if ( useful(task,Global_best) ) {
    best=solveTask(task,Global_best);
    if ( best < Global_best )
#pragma omp critical
        if ( best < Global_best )
          Global_best = best;
  }
```

The reader may notice that the only sensible part of the code consists in the critical zone, necessary to update the `Global_best` variable.

2) Tasks allocation in a parallel region: A parallel region is executed by all the processors. The user is explicitly in charge of distributing the load among the processors.

We showed in [10] how to emulate the main parallel *for* loop schedule with parallel regions, and proposed other adapted load balancing strategies to solve the Langford problem in parallel. The good results obtained for this study encouraged us to carry on in this direction.

For the specific problem of constructing optimal Golomb rulers, we first noted that most of the initially useful tasks have actually became useless when they have to be treated (see the experimental results in next section). This phenomenon induces that, when a processor becomes idle, it has to take several tasks before having an *interesting* one. This creates a high concurrency for the tasks allocation which may considerably reduce computation efficiency.

Thus, two different load balancing strategies using parallel regions are proposed :

- A servor initiated one first divides the tasks into tasks queues in the static modulo manner. Every processor begins with its queue. When a processor becomes idle it chooses a non empty queue and takes tasks in it, one by one, in a *while* manner, or by directly taking half of this queue.
- The client-servor load balancing strategy shares the index of the next available task between the processors. In order to improve the "one by one" dynamic schedule, processors take as many tasks as necessary to find a useful one: a lock is set on the $next$ variable and tasks are consulted in a *while* manner; then the lock can be unset. The advantage of this strategy is to reduce the number of times a lock is set, but the locks are set for a longer period.

D. A parallel version using a Message Passing Interface

Message passing is a programming paradigm used widely on parallel computers, especially with distributed memory. The Message Passing Interface (MPI) is a standard approach for message passing programming [11]. This standard defines the user interface and functionality for a large number of message-passing capabilities and with a large degree of portability.

As it represents the simplest solution for the user, the tasks allocation is done by a client/server scheme. The skeleton of the MPI program can now be introduced. Note that all the processors generate the possible tasks and then that the servor distributes them.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nbp);
MPI_Comm_rank(MPI_COMM_WORLD, &p);
if ( p == 0 ) { /* server */
  nbTasks = generateTasks(n, max,k);
  global_best = initial_limit;
  initialSends();
  while ( (nextTask < nbTasks)) {
    /*  get the result */
    MPI_Recv(&noClient, 1,
    MPI_INT, MPI_ANY_SOURCE,
    1, MPI_COMM_WORLD, &ierr);
    MPI_Recv(&best, 1, MPI_INT,
    noClient, 2, MPI_COMM_WORLD, &ierr);
    if (best < global_best)
      global_best = best ;

    /* send the next task */
    nextTask = nestUseful(nextTask);
    MPI_Send(&nextTask, 1, MPI_INT,
    noClient, 0, MPI_COMM_WORLD);
    MPI_Send(&global_best, 1, MPI_INT,
    noClient, 0, MPI_COMM_WORLD);
  } /* end while */
  finalRecieves();
  MPI_Finalize();

} /* end of server */
else { /* client */
  nbTasks = generateTasks(n, max,k);
  best = initial_limit;
  MPI_Recv(&task, 1, MPI_INT, 0, 0,
  MPI_COMM_WORLD, &ierr);
  while ( task < nbTasks ) {
    best = solveTask(task,best);

    /* send the result */
    MPI_Send(&p, 1, MPI_INT, 0,
    1, MPI_COMM_WORLD);
    MPI_Send(&best, 1, MPI_INT, 0,
    2, MPI_COMM_WORLD);

    /* receive next task */
    MPI_Recv(&task, 1, MPI_INT, 0,
    0, MPI_COMM_WORLD, &ierr);
    MPI_Recv(&best, 1, MPI_INT, 0,
    0, MPI_COMM_WORLD, &ierr);
  } /* fin du while */
  MPI_Finalize();

} /* end of client */
```

IV. Experimental results

As evoked in the previous sections, the tested algorithm takes advantage of a full cooperation between the tasks, through the processors that share the information of the best known length.

The experimental evaluation we present here was led using a Silicon Graphics Origin'3800 with 512 R14K 500 MHz processors. This study was limited to 32 processors to experiment the effects of granularity and of the different load balancing strategies, and to 128 for constructing further rulers.

A. Granularity

Considering a given problem with a given value of $initial\_limit$, the choice of the depthlevel may influence computational time, as shown by the following example, where G(13) was computed with $initial\_limit = 200$ on 32 processors (see figure 4).

- The depthlevel must not be too small because a good load balancing is impossible if the average number of tasks per processor is not important enough. For example, with depthlevel 1, only 45 tasks were generated: this created a high irregularity (one of the processors finishing in 16.3 seconds and another one in 696.2)
- If the number of tasks becomes too important, the processors may spend some more time in racing to have new tasks instead of computing the tasks themselves.

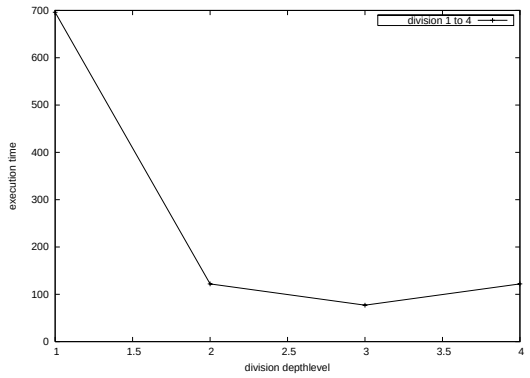


Fig. 4. G(13), $initial\_limit$=200, influence of the granularity

B. Load balance

Because of the tasks irregularity, the OpenMP static and static modulo schedules can't have good results: for example for the computation of G(13) using 32 processors with a limit[2] of 147 and dividing the search tree with depthlevel 3, one finishes in about 60 seconds and another one in about 140).

Table I presents the results of the computation of $G(14)$ using 4 to 32 processors with a limit of 181 and dividing the search tree with depthlevel 3, comparing the OpenMP dynamic schedule on a parallel loop (*Dyn*) with the OpenMP servor initiated and client servor parallel region schedules (*SI* and *CS*), and the MPI client servor. Note that the sequential execution takes 31450 seconds.

TABLE I
Execution times for $G(14)$ in seconds

	OpenMP			MPI
Procs	**Dyn**	**SI**	**CS**	**CS**
4	7643.7	7501.1	10951.9	10258.1
8	3852.2	3814.6	3971.5	4582.7
12	2714.1	2679.9	2831.3	2881.5
16	1972.0	1997.9	2008.3	2216.5
24	1466.0	1441.8	1521.4	1535.9
32	1147.7	1038.1	1064.1	1115.0

Using a parallel *for* loop with a dynamic schedule gives very good results (with an efficiency of about 93%), however adapted the other OpenMP strategies are.

The MPI code gives good results too, with an efficiency of about 88% while the optimal efficiency is equal to 91% taking the server into account. The penalty induced by the server is less important when the number of processors increases, and may become tiny with a very large number of processors.

C. Further results

Using a great number of processors makes it possible to construct optimal Golomb rulers for larger instances. For example, 32 processors gave the result for G(15) (with $initial\_limit$=200 and depthlevel=3) in 4h40, and even in 1h25 with 128 processors.

D. How the OpenMP experiments were led

To conclude, a remark has to be made on the computational irregularity observed for the OpenMP versions of the program.

Each computation has been repeated 20 times. For example, $G(13)$ with the dynamic *for* loop and 2 processors was computed in 991-995 seconds 17 times and in 1150-1540 seconds 3 times. When the number of processors

[2]The limit is set to 200 and a preliminary treatment constructs a first solution of length 147 instantly.

increases, this instability is observed more often. Thus, $G(13)$ was computed on 16 processors using the modulo *for* loop in about 180 seconds only 1 time in 20 tries, and the relative standard deviation was about 135% (the values went from 180 to 1110 seconds and even two computations didn't finish in the 30 minutes allowed).

Such an irregularity was observed on all the parallel machines we used (an IBM SP Power 3 NH2 and a Sunfire 6800)

As the architecture is shared by different users, the application has no guaranty about the memory allocation. On the other hand their is no way to know where a processor allocates its own memory; moreover shared memory, allocated in a particular place, is *near* some of the processors and *far* from other ones, whose memory access times may become very bad. The effects are increased by the fact that our algorithm uses much memory and is based on a very large number of memory accesses, in private memory as well as in shared one. This is the reason why we had to do many computations for each experiment, and we only retained the best value (which corresponds to the "good" memory allocation case).

V. Conclusion and perspectives

In this paper, the parallel construction of the optimal Golomb rulers has been studied using a tree search algorithm. The application has been written in C using OpenMP or MPI. The key advantage of OpenMP is to offer a user-friendly tool to parallelize the sequential algorithm: with minimum changes to this algorithm, it has been possible to design an efficient parallel version. The experiments provided on SGI'3800 show that the parallel *for* loop with the dynamic schedule provided by OpenMP is an effective solution. The use of OpenMP parallel region or MPI are also possible, but the programmer has to make some efforts and change some of the code structure to design an efficient parallel application.

A major drawback highlighted by the experiments is that with OpenMP the execution times for the same problem can be very different from one execution to another. It is probably due to the memory management in a multi-user environment.

The main perspective of this work is to write a hybrid solution based on OpenMP and MPI to solve the next instances more quickly and compute open ones. As the computation effort grows geometrically with the number of marks, the execution on a large cluster of SMP (Symmetric Multi-Processor) should be considered as an interesting alternative to solve one of the first open instances in an acceptable range of time. Inside a SMP node, the parallelism should be managed in a shared memory by OpenMP. Outside the nodes, a message passing solution would enable the management of the load between the nodes.

Acknowledgments

This work was partly supported by "Romeo"[3], the high performance computing center of the University of Reims Champagne-Ardenne and the "Centre Informatique National de l'Enseignement Superieur"[4] (CINES), France.

References

[1] Z. Habbas, M. Krajecki, and D. Singer, "Parallel resolution of csp with openmp," in *Proceedings of the second European Workshop on OpenMP*, Edinburgh, Scotland, 2000, pp. 1–8.

[2] M. Gardner, *Mathematical Games*. Scientific American, 1972.

[3] W. Rankin, "Optimal golomb rulers: An exhaustive parallel search implementation," Master's thesis, Duke University, 1993.

[4] W. Bloom and S. Golomb, "Applications of numbered unidirected graphs," in *Proceedings of the IEEE, 65second European Workshop on OpenMP*, Edinburgh, Scotland, 1977, pp. 562–570.

[5] B. Smith, K. Stergiou, and T. Walsh, "Modelling the golomb ruler problem," 1999.

[6] S. Soliday, A. Homaifar, and G. Lebby, "Genetic algorithm approach to the search for golomb rulers," in *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman, Ed. San Francisco, CA: Morgan Kaufmann, 1995, pp. 528–535.

[7] Z. Habbas, M. Krajecki, and D. Singer, "Parallelizing Combinatorial Search in Shared Memory," in *Proceedings of the fourth European Workshop on OpenMP*, Roma, Italy, 2002.

[8] *OpenMP C and C++ Application Program Interface*, OpenMP Architecture Review Board, Oct. 1997, http://www.openmp.org.

[9] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[10] C. Jaillet and M. Krajecki, "Solving the langford problem in parallel," in *Proceedings of ISPDC'04, International Symposium on Distributed and Parallel Computing*, Cork (Ireland), 2004.

[11] P. Pacheco, *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[3] `http://www.univ-reims.fr/Calculateur`

[4] `http://www.cines.fr`

Session 3: OpenMP implementation issues

Chair: Dieter an Mey, Aachen University, Germany

- Y. Hotta, M. Sato, Y. Nakajima, and Y. Ojima, OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor
- S. Karlsson, A Portable and Efficient Thread Library for OpenMP
- H. Matsuba and Y. Ishikawa, OpenMP on Distributed Memory Computers with FDSM Distributed Shared Memory

OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor

Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, Yoshinori Ojima
Graduate School of Systems & Information Engineering,
University of Tsukuba
{hotta,msato,yoshihiro,ojima}@hpcs.is.tsukuba.ac.jp

Abstract

CMP (Chip Multiprocessor) is a promising processor architecture, not only for high performance but also for reducing power and energy consumption in embedded applications. We have implemented an OpenMP compiler for an embedded Renesas M32R chip multiprocessor as a parallel programming environment. In this paper, we report the preliminary performance of OpenMP benchmarks, including scientific and multimedia applications on the M32R CMP. We found that OpenMP allows users to easily obtain reasonable performance improvement using multiple CPUs in the CMP with just a few directives inserted. Also, we discuss the possibility of OpenMP thread run-time scheduling and some compilation techniques for power-aware computing on the CMP.

1. Introduction

Recently, many processor developers have announced a chip multiprocessor (CMP) architecture, especially for embedded applications. ARM offers MPCore [1], which is a CMP with four processors in a single chip. Even in the high-performance computing field, Intel announced that they will change their high-end processor s road map to CMP due to the chip s high power dissipation and low clock frequency [8]. As more transistors areintegrated onto a bigger die, the CMP will become a promising alternative to the complex superscalar microprocessor that is commonly used today. While there seems to be a trend toward CPUs with wider instruction issues as well as support for a larger amount of speculative execution, a CMP composed of simpler processors may exploit thread-level parallelism efficiently. For such CMPs, parallel (or multi-threaded) programming is required. OpenMP is an easy-to-use simple parallel programming environment that makes use of the multiple CPUs in the CMP.

In this paper, we report an implementation of OpenMP and its preliminary performance on a Renesas M32R chip multiprocessor, which is intended for embedded systems. M3T-M32700UT is a system designed to develop several embedded applications using M32R CMP. M3T-M32700UT has several devices, such as Ethernet, LCD, and a video camera, and it runs on the embedded Linux.

One of our research interests is power-aware computing using a CMP. Recently, there has been tremendous interest in power-aware computing for mobile embedded systems such as PDAs and cellular phones. Even in high-performance parallel computing systems, it is very important to reduce the power consumption for cooling and high-density packaging. Our recent research [2] shows that a cluster of low-power processors can obtain better performance than a single high-performance processor in terms of the power-performance ratio. CMP allows us to exploit both high performance and low power since it can achieve high performance while keeping a low clock frequency by having multiple processors on a single chip.

An interesting feature of OpenMP is that the execution of OpenMP programs is independent of the number of actual physical processors. This feature can be a useful tradeoff between application performance and power consumption in the CMP. For example, when an application is required to reduce the power consumption, which may cause the application to run slowly, we can run the OpenMP program at low power by stopping the power supply to some of processors.

The roles of OpenMP for embedded CMP are as follows:

- Providing a portable parallel programming environment. OpenMP provides a standard API in C/C++ and Fortran for a shared memory machine used for embedded applications.
- Providing an easy to use parallel programming model. The original sequential program may be parallelized easily by inserting OpenMP directives. Actually, the POSIX thread library is often used to develop

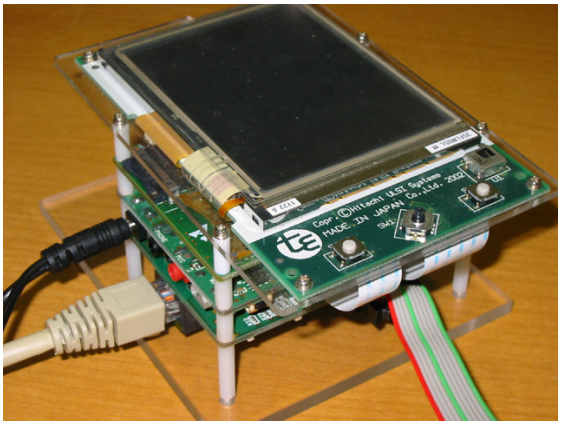


Figure 1. Photo of M3T-M32700UT

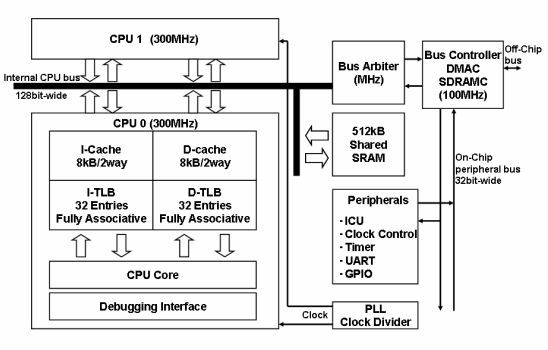


Figure 2. Block Diagram of Renesas M32R CMP (M32700)

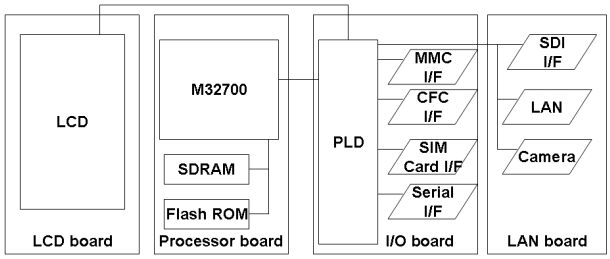


Figure 3. M3T-M32700UT block diagram

feature	spec
CPU	Renesas M32R CMP (M32700)
clock	300MHz (in this paper at 200MHz)
cache I/D	8KB/8KB 2way
Memory	16MB SDRAM / FLASH:4MB
LAN	100Base-TX
USB	Host2.0
module	CF/MMC/eTRON slot, LCD, AR camera
size	60mm * 60mm

Table 1. Specification of M3T32700UT

embedded applications. An OpenMP-style programming model can reduce the cost of parallel programming by eliminating tedious codes for managing and synchronizing threads.

- Exploiting parallelism on embedded multimedia applications. Time-consuming multimedia applications often have much parallelism so that the parallelization makes these applications faster.
- Enabling flexible run-time thread scheduling for real-time and power-aware computing. As described above, sophisticated run-time thread scheduling may allow a tradeoff between application performance and power consumption by controlling the CPU status in the CMP.
- New features of OpenMP will be possible to extend OpenMP for real-time embedded applications.

In this paper, we discuss some techniques for reducing power and energy consumption using OpenMP. Reducing power and energy is important for embedded systems to make a compact package while maintaining performance. The rest of the paper is organized as follows. The specification of our target system M3T-M32700UT is presented in the next section. To implement an OpenMP compiler, we used the Omni OpenMP compiler, which is described in Section 3. Section 4 presents the preliminary performance evaluation using some scientific and multimedia benchmarks. Section 5 discusses the OpenMP issue for power-aware computing in CMP. Finally, we present our concluding remarks and future work in Section 6.

2. M3T-M32700UT: Renesas M32R Chip Multiprocessor System

M3T-M32700UT is a system that consists of an M32R CMP processor [3], LCD, network interface, etc. The M32R processor is a product of Renesas Technology and is a very low-power processor for embedded systems. Table 1 shows the specification of the M32R CMP processor. Figure 1 shows a photo of the M3T-M32700UT.

The M32R32700 single-chip multiprocessor contains two M32R 32-b CPU cores and 512 KB of shared SRAM, and was designed for embedded microcontrollers and a system-on-chip (SoC) core. Since the software for an embedded system has increased in complexity, embedded processors are required to increase performance. At the same time, low power dissipation is still a key feature for battery-operated applications.

This chip is fabricated in a 0.15 μm CMOS process. Figure 2 shows a block diagram of the chip. Two CPU cores and the shared SRAM are connected via a 128-bit bus. The pipeline is a 7-stage pipeline. The CPU core has 2-way set-associated I/D cache memory, and 32-bit full-associated memory. The power consumption is about 800 mW (in the latest M32R processor, CPU clock: 400 MHz, bus clock: 100 MHz) Table 1 shows the M3T-M32700UT specification. This system has an LCD, network interface, camera, etc., for developing applications. As an operating system, Linux/M32R [5] is the ported Linux for the M32R. The Linux/M32R supports both the 2.4 and 2.6 kernel. Several device drivers (e.g., LAN, CF, USB, MTD) have already been ported. This system is designed for development of an embedded application. We used this system for the performance evaluation of OpenMP applications.

3. OpenMP Implementation

To implement OpenMP on the M32R CMP, we used the Omni OpenMP compiler [7]. The Omni OpenMP compiler is a translator that takes OpenMP programs as input to generate a multi-threaded C program with run-time library calls. The generated program is compiled by the native cross compiler and linked with the run-time library. The run-time library includes microtasking and synchronization primitives on top of the different thread libraries, which includes POSIX threads. Since Linux/M32R supports POSIX threads, it was very easy to port the run-time library to the system.

Table 2 shows the overhead of Synchbench using the EPCC microbenchmark. For reference, we also measured the performance on a Pentium-based SMP platform (Pentium III 550 MHz). The overhead of this result, we found dramatically difference between M32R CMP (2 processors) and Pentium III (2 processors). In particularly "CRITICAL" and "LOCK/UNLOCK" spends over few hundreds of times time. On the other hand, we found little difference in "FOR" and "BARRIER" overhead. But in all case, we found the ratio of each processors overhead is large than the difference of processors clock frequency. So, this result means as compared with high performance processor, M32R CMP has large overhead in case of using OpenMP.

processor (2 proc.)	M32R	PenIII	ratio
PARALLEL	843.4	4.7	179.4
FOR	17.7	1.3	13.6
PARALLEL FOR	857.8	5.2	164.9
BARRIER	13.0	0.9	14.4
SINGLE	58.9	1.4	42.0
CRITICAL	313.8	0.7	448.3
LOCK/UNLOCK	313.5	0.7	447.8
ORDERED	10.3	0.4	25.7
ATOMIC	295.4	0.9	328.2
REDUCTION	855.7	6.8	125.8

Table 2. Results of EPCC micro benchmark [micro sec]

	single	2 proc.	speedup
Laplace	109.2	56.7	1.9
IS CLASS S	0.31	0.19	1.63

Table 3. The execution time and speedup of benchmarks [sec]

4. Preliminary Performance Evaluation

For the performance evaluation, we used some benchmarks including Laplace, NPB (NAS parallel benchmarks)3.1 [6] and MediaBench [4]. NPB and Laplace are taken as scientific workloads, MediaBench is a multimedia application. We used an m32r-linux-gcc compiler (gcc3.2 based) as a backend.

4.1 Laplace and NPB IS

The Laplace benchmark is a simple Laplace equation solver with 4-point stencil operation (512*512 and 20 iterations). NPB IS is an integer sort program taken from NPB 3.1. The size of the benchmark is class S. Each benchmark has different characteristics: Laplace is a computation-intensive program and NPB IS is a memory-intensive program. Table 3 shows the result of these benchmarks. We found that performance dramatically improved by using two processors. In the Laplace benchmark, we achieved a speedup that was almost double. In the IS benchmark, however, the speedup was limited to 1.6. Therefore, we found less speedup in the memory-intensive program than in the computation-intensive program.

4.2 MediaBench

For evaluating the performance of multimedia applications, which are the main targets of embedded systems,

function	single	2 proc.	speedup
motion-estimation	1.823	1.009	1.81
predict	0.001	0.001	0.95
dct-type-estimation	0.018	0.019	0.97
transform	0.374	0.018	1.99
putpict	0.231	0.223	1.04
iquant	0.018	0.020	0.92
itransform	0.038	0.050	0.76
calcSNR	0.025	0.026	0.95
Total	2.762	1.55	1.65

Table 4. The execution time of mpeg2encode on Pentium III SMP (550Hz) [sec]

function	single	2 proc.	speedup
motion-estimation	9.03	4.94	1.83
predict	0.12	0.12	1.01
dct-type-estimation	0.46	0.53	0.87
transform	102.62	73.06	1.40
putpict	2.32	3.81	0.62
iquant	0.13	0.13	1.00
itransform	0.25	0.26	0.99
calcSNR	18.61	11.59	1.60
Total	131.25	94.48	1.41

Table 5. result of mpeg2encode

we used MediaBench. MediaBench is a benchmark for embedded multimedia applications. This benchmark contains applications collected from available image processing, communications and DSP applications. It includes MPEG, JPG, GSM, G.721, PGP, and others. While all of MediaBench may be parallelized by OpenMP, we used the MPEG2 encode, which is a very suitable benchmark for evaluating the benefit of OpenMP on M3T-M32700UT. We parallelized the MPEG2 encode by OpenMP. MPEG is the dominant standard for high-quality digital video transmission. The important computing kernel is a discrete cosine transform for coding and an inverse transform for decoding. The two applications used are mpeg2enc and mpeg2dec for encoding and decoding, respectively. In this paper, we showed the performance of mpeg2enc. Figure 4 shows the MPEG2 encoding stages. This benchmark contains motion-estimation, predict, dct-type-estimation, transform, putpict, iquant, itransform, and calcSNR functions. The motion-estimation function is the calculation of a motion vector of the luminance frame in each macroblock. The basic premise of motion estimation is that, in most cases, consecutive video frames will be similar except for the changes induced by objects moving within the frames. predict predicts the motion compensation. dct-type-estimation is the selection DCT mode from a frame or field DCT. transform is the discrete cosine transformation (DCT) using DCT mode. putpict is the quantization of DCTed blocks and output of bit stream code. iquant is the inverse quantization to the quantized blocks. itransform is the inverse DCT to the dequantized blocks for reconstructions. calcSNR is the calculation of the signal to noise Ratio (SNR). The input data size is four frames of 352*240 in MediaBench.

For reference, we also measured the performance on a Pentium-based SMP platform (Pentium III 550 MHz). Table 4 shows the result of mpeg2enc on this system. We found that the motion-estimation took a large part of the execution time, so we parallelized motion-estimation and transform. For example, we show the code transformation on motion-estimation.

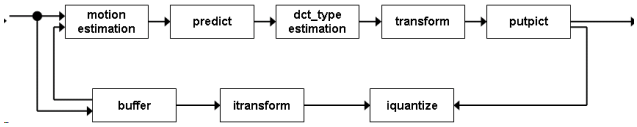


Figure 4. Stages of MPEG2 encode

```
/*loop through all macro-blocks of the picture*/
#pragma omp parallel private(i,j,myk)
{
 #pragma omp for
 for (j=0; j<height2; j+=16)
 {
  for (i=0; i<width; i+=16)
  {
        ... loop body ...
  }
 }
}
```

Table 5 shows the execution time of each function of mpeg2encode on the M32R CMP. The performance is low mainly because of its low clock frequency. In the case of the single processor, the ratio of execution time of motion-estimation, transform, and calcSNR changes compared to that in the Pentium III. This is because the M32R does not have a floating point unit. The execution time of functions that contain many floating point calculation becomes long. In the M32R CMP, we also parallelized calcSNR for performance improvement. For the two-processor execution, we found a performance improvement of the parallel version of about 30% in total execution time. While the performance improvement of the motion-estimation and the calcSNR functions is reasonable, the speedup in the transform function is limited to 1.40. We found that the transform function spends almost as much time as the DCT transformation calculation (over 99%) in a function. we guess that the transform cannot achieve more speedup than the other function because M32R has only L1 small cache.

It should be noted that OpenMP enables us to obtain performance improvement by adding only a small amount of

code. We believe that OpenMP helps the programmer to parallelize the embedded application very easily for an embedded CMP.

5. OpenMP on CMP for Power-Aware Computing

We are interested in power-aware computing using CMP for embedded applications. In this section, we describe several issues regarding compilation technique and OpenMP for power-aware computing.

5.1 Processor run-time scheduling for real-time applications

The typical requirement in real-time applications is to execute a reserved job within a certain period. Although parallel execution runs a program faster than sequential execution does, it is not necessary to execute in parallel with respect to power efficiency. Even though it may take a long time to meet the deadline, the program can run on one CPU with other CPUs in the standby mode. For such run-time thread scheduling, co-scheduling with OpenMP run-time and kernel thread scheduling on the CMP will be required.

In our experiment, we measured the power consumption. The power consumption of the system was 3 W all the time (we measured DC power including CPU, LCD, network and some devices). Unfortunately, we did not find much difference between the OpenMP execution and the sequential execution, because most of the power was used for devices such as the LCD and the network, and the power consumption of the CPUs is only on the order of a few hundred mW.

In addition, there is no kernel support to put the CPU in standby mode. In the near future, we will investigate kernel support and OpenMP thread scheduling for power-aware computing.

5.2 Compiler optimization techniques for low-power computing

Much compiler research has already been done in low-power embedded computing. In embedded applications, not only power consumption but also energy consumption is very important. Power consumption is significant for cooling and packaging, energy consumption is significant for battery-life.

Valluri [10] investigated the impact of using a different level of optimization in the compiler for system power and energy. This research found that most compiler optimizations also benefit the energy by reducing the number of operations, such as dead-code elimination and constant propagation. Higher IPC makes power higher, but the execution time reduces dramatically by compiler optimization. The ratio of execution time reduction is dramatically larger than that of power addition. So we can reduce the energy by compiler optimization. Steinke et al. [9] applied register pipelining to reduce power and energy. Their technique is to replace the memory load with register read (i.e., use cheaper instructions).

According to our previous research, cache optimization such as cache blocking is more effective in reducing the energy in a low-power embedded processor than in a high-performance processor. In a system using embedded processor power consumption in the case of cache hit is smaller than that of cache miss hit. And almost embedded processor has only L1 cache memory, so program can cause cache miss hit and memory bandwidth is small. From that reason, performance is dramatically improved by cache-optimization in embedded processor than conventional high performance processors. While the effect of execution time in case of cache-blocking of high performance processor is about 2 times faster than non-blocking, that of embedded processor is over 9 times faster than non-blocking. As a result, we can reduce energy dissipation dramatically in case of cache-blocking optimization.

So, the cache-blocking technique and memory optimization by the compiler is very effective for power-aware computing. The OpenMP compiler might help code transformations in the above compiler optimization for reducing power consumption.

6. Concluding Remarks and Future Work

We have implemented an OpenMP compiler on an embedded M32R CMP system, and have evaluated its preliminary performance using scientific and multimedia benchmarks. OpenMP is very useful for parallelizing embedded applications as well as scientific applications on the CMP. We found that reasonable improvement can easily be obtained just by inserting OpenMP directives. We have proposed the possibility of OpenMP run-time scheduling for a tradeoff between performance and power in real-time embedded applications in power-aware computing. The most current OpenMP compilers and run-time systems are designed for shared memory multiprocessors. If CMP has a sophisticated mechanism for fast synchronization on a chip, an OpenMP implementation could be redesigned to make use of such fast synchronization and low-latency communications between threads in the run-time system and compiler. Flexible thread run-time scheduling is also an interesting design issue of OpenMP to exploit nested dynamic parallelism using fast synchronization of the CMP. For reducing power consumption, CMP architecture is very effective. In CMP, OpenMP has a large impact for the application engineer, and with power-effective scheduling using

OpenMP, we can greatly reduce power and energy.

Acknowledgment

We would like to thank Renesas Technology Corp. for providing M3T-M32700UT for this research. This research is partly supported by the Grant-in-Aid of Ministry of Education, Culture, Sports, Science and Technology in Japan, No. 14208026, 2002-2004.

References

[1] ARM MPCore. http://www.arm.com/.

[2] Y. Hotta, M. Sato, T. Boku, D. Takahashi, and C. Takahashi. Measurement and chracterization of power consumption of microprocessors for power-aware computing. In *CoolChips7*, April 2004.

[3] S. Kaneko et al. A 600MHz Single-Chip Multiprocessor with 4.8GB/s Internal Shared Pipelined Bus and 512kB Internal Memory. In *2003 International Solid-State Circuits Conference*, volume 1, pages 254–255, September 2003.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[5] Linux/M32R Project. http://www.linux-m32r.org/.

[6] NAS Parallel Benchmarks. http://www.nas.nasa.org/NAS/NPB.

[7] OpenMP Forum. http://www.openmp.org.

[8] R. Sasanka, S. V.Adve, Y.-K. Chen, and E. Debes. The energy efficiency of cmp vs smt for multimedia workloads. In *ISC'04*, July 2004.

[9] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *the International Symposium on System Synthesis*, October 2002.

[10] M. Valluri and L. John. Is Compiling for Performance == Compiling for Power? In *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, January 2001.

A Portable and Efficient Thread Library for OpenMP

Sven Karlsson
Department of Microelectronics and information Technology
Royal Institute of Technology, KTH, Sweden
email: sven.karlsson@imit.kth.se

Abstract

The design of a portable, yet efficient, thread library, called Balder Threads, is discussed in this paper. The library is used within Balder, a run-time library for OpenMP 2.0. The thread library is evaluated using the EPCC micro-benchmarks and measuring the overheads for the entire Balder OpenMP run-time library. The overheads, using Balder Threads, are found to be an order of an magnitude smaller than when using POSIX primitives. The overheads achieved are comparable to those of a commercial system from Intel.

1. Introduction

A compilation system for OpenMP typically consists of an OpenMP aware compiler and a run-time library system. The run-time library not only provides the intrinsic OpenMP functions as described by the OpenMP specification but also generally provides important support functions used by the compiler for thread creation and thread synchronization. The efficiency of these support functions directly influences the performance of OpenMP constructs.

One OpenMP run-time library is the *Balder* library which is targeted by the OdinMP compiler [3, 4, 5]. The library and the compiler fully support OpenMP 2.0 including nested parallelism [6]. The Balder library has a modular design and has several sub-libraries. One such sub-library is an efficient portable thread library called *Balder Threads*. In this paper the functionality of the primitives in Balder Threads is discussed as well as the design of the library. It is shown, using the EPCC micro-benchmarks [1], that the Balder Threads library provides an order of magnitude better performance than POSIX primitives for a number of common OpenMP constructs while still being highly portable.

The existing non-commercial run-time libraries either use specialized non-portable thread libraries or POSIX primitives which incur a high overhead. The, in this paper, proposed system provides portability while still being efficient.

The rest of the paper is organized as follows. The Balder and Balder Threads libraries are discussed in section 2. Experimental results are presented in section 3 while the paper is concluded in section 4.

2. Balder and Balder Threads

Balder is an OpenMP run-time library supporting OpenMP 2.0 [4]. The library supports SMPs as well as clusters of SMPs and is designed as a layered system, see figure 1.

The library consists of three sub-libraries. The sub-libraries are *Balder Messages* which is a messaging library used when targeting a cluster, *Balder Oslib* which provide an operation system independent interface to operating system resources, and finally Balder Threads which provides thread primitives. The actual OpenMP functionality is implemented in a layer which is built on-top of the mentioned three libraries. The OpenMP layer also includes a software distribution shared memory system, software DSM system, which provides a shared address space when targeting a cluster. The software DSM system, as well as the messaging layer, is completely inactive when using a single SMP node and will not inflict any overhead in that case. I will not touch on clustering issues in this paper.

The entire run-time library is written in C using standardized operating system APIs to achieve portability. Some operation system services needed by the library, such as the virtual memory system, do not have standardized APIs but the Balder Oslib provides a virtualization layer to these services.

Balder has been ported to several different variants of Unix as well as various revisions of Windows. It has also been ported to several processor architectures.

The focus of this paper is the Balder Threads library which provides threading primitives. In short the following primitives are provided:

- Thread creation and thread destruction.
- Thread synchronization through monitors.
- Barriers.
- Work-queues.

Internally, the OpenMP layer manages a pool of threads. This means that threads are only created when the thread pool is empty and so thread creation need not be very efficient. The other primitives need however be very efficient. The thread synchronization primitives are for instance used internally by Balder for mutual exclusion and also for the intrinsic OpenMP lock functions and the critical construct.

The Balder Threads library uses the POSIX thread API for thread creation, thread destruction and synchronization [2]. The POSIX thread API is a general thread API and has, as can be seen in section 3, a fairly high overhead. To improve on the overheads, Balder Threads implements a layer, on-top of functions from the POSIX thread API, which performs synchronization using processor architecture dependant primitives such as test-and-set and fetch-and-add. Functions from the POSIX thread API are then used to block threads. The processor dependant primitives are implemented as short assembly language snippets embedded in pre-processor macros making it possible to implement the synchronization primitives in portable C code. Only the processor dependant macros need to be implemented to port Balder Threads to a new processor architecture variant.

The basic form of synchronization in the Balder Threads library is spin-locks based on test-test-and-set. The target for the Balder library is SMPs up to roughly 8 CPUs and so more elaborate synchronization primitives was deemed not needed. Experiments conducted shows that the test-test-and-set procedure used is reasonable even up to as much as 64 CPUs.

As mentioned, the Balder Threads library uses a number of macros to make use of processor dependant primitives. The set of primitives supported through the macros are:

- Test-and-set. This is, as outlined above, the basic synchronization primitive and is the only synchronization primitive which is required by the library to be implemented in a port.
- Memory fences. The library makes use of memory fences if needed by the underlaying processor architecture. The library assumes that the memory fences control both read and write operations and inserts fences before reading, and after writing, shared variables used for synchronization. The memory fence macros are blank for architectures whose memory consistency models do not require memory fences.
- Fetch-and-add. The fetch-and-add is used through out the library to reduce synchronization cost for various shared variables that are used as counters. The primitive is in particular used to reduce the overhead of barriers and work-queues.

In addition to these macros there are macros, required by the library, which define the layout of stack frames and define how functions are called. These macros are used to implement the parallel construct. Required macros also define the size and layout of cache lines. These macros are used to lay out spin-lock variables so that each spin-lock resides in its own cache

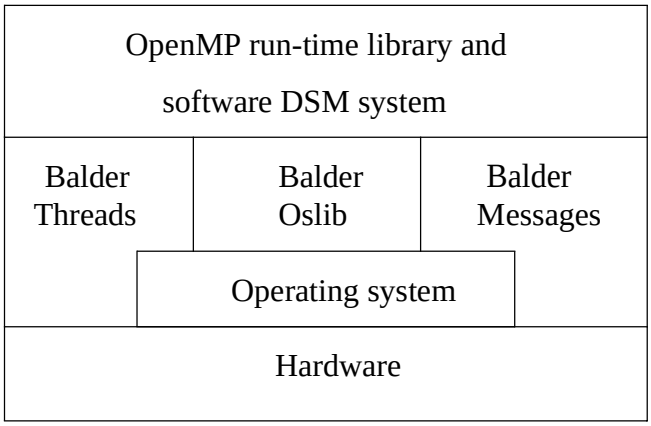


Figure 1. Overview of the design of the Balder library.

Table 1. Overheads in microseconds of common OpenMP constructs.

OpenMP Construct	Intel compiler	Balder with OdinMP without architecture support	Balder with OdinMP and architecture support
Parallel construct	1.43 +/- 0.11	27.15 +/- 1.35	2.91 +/- 0.15
For construct	0.79 +/- 0.17	10.59 +/- 0.87	2.93 +/- 0.30
Barrier construct	0.48 +/- 0.19	11.14 +/- 0.29	0.49 +/- 0.12
Lock and unlock primitives	0.48 +/- 0.33	6.90 +/- 0.35	0.47 +/- 0.12

line so as to make sure coherency traffic is kept at a minimum. POSIX primitives are used as efficiently as possible for synchronization if no macros are provided.

The test-test-and-set variant used in Balder Threads performs a time-out if a spin-lock has not been released within certain time period of busy-waiting. The actual time-out period is set to a time roughly equal to two context switches at system startup through run-time measurements on POSIX thread API primitives but the time-out period can also be fine-tuned through command-line options. The purpose for the time-out period is to avoid excessive busy-waiting in systems where several processes compete for processor resources.

A pseudo code describing the test-test-and-set procedure and the monitor entry and leave functions is outlined in figure 2.

3. Experimental results

A few experiments have been conducted so as to assess the performance of the thread library. A dual Pentium-III workstation running Linux version 2.4.25 was used as experimental platform. The processors were running at a clock rate of 1 GHz.

The EPCC micro-benchmark suite was used in the experiments [1]. The benchmarks were compiled with OdinMP version 0.284.1 and GCC 3.3.4. The Balder library version 0.105.1 was used and was also compiled with GCC 3.3.4.

For comparison, the same set of benchmarks were compiled with the Intel C/C++ compiler version 8.0 and run on the experimental platform. The Intel compiler supports OpenMP. The Balder library cannot currently be compiled with the Intel compiler. The highest available optimization level was used in both compilation systems.

The overheads in microseconds for the most common OpenMP constructs are summarized in table 1. The overheads are presented with their 95% confidence interval. The overheads in the first three rows are for one single parallel region, parallel for-loop, and barrier respectively. The lock and unlock primitives row is the overhead of setting and then releasing a single lock once. Two versions of the Balder library were used. One with full architecture support including test-and-set and fetch-and-add operations and one where all architecture support is disabled forcing POSIX primitives to be used for all synchronization.

The overheads are reduced by as much as an order of magnitude when using architecture support when compared to pure POSIX primitives. The overheads for synchronization primitives are as good as for the commercial Intel compiler while higher for parallel for-loops and creation of parallel regions. The reason lies in the way the OdinMP compiler generates code and how parallel regions are handled by the run-time library.

4. Conclusions

In this paper, the thread library in the Balder OpenMP run-time is presented. The thread library is called Balder Threads and is highly portable. The paper discusses the design of Balder in general and Balder Threads in particular. The performance of the thread primitives are evaluated using the EPCC micro-benchmark suite and is found to be an order of magnitude better than that of POSIX primitives. The performance is also found to be comparable to that of a commercial compilation system from Intel.

```
int test_test_and_set(spin_lock_type* spin_lock_variable)
{
  do
  {
    MFENCE;                    /* MFENCE is a macro defining
                                  a memory fence operation. */
    while(*spin_lock_variable==TAKEN)
    {
      MFENCE;                  /* Wait and perform memory fences
                                  until the spin-lock variable is
                                  free. */
      if (timeout occured)
      {
        return 1;
      }
    }
  } while (TEST_AND_SET(spin_lock_variable)==TAKEN);
                               /* TEST_AND_SET is a macro defining
                                  a test_and_set primitive. */
  return 0;
}

void enter_monitor(monitor_variable_type* monitor)
{
  while (test_test_and_set(&monitor->spin_lock_variable))
  {
                               /* A time-out has occurred. The
                                  thread will be blocked using
                                  POSIX functions and condition
                                  variables. */
    block thread;

  }
}

void leave_monitor(monitor_variable_type* monitor)
{
  monitor->spin_lock_variable=FREE;
  MFENCE;
  if (threads are blocked)
  {
    unblock at least one thread;
  }
}
```

Figure 2. Pseduo code for monitor entry and monitor leave.

Acknowledgements

The work in this paper has been partly financed by the European Commission in the Intone project under contract number IST-1999-20252.

References

[1] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sept. 1999, pp. 99-105

[2] IEEE, *IEEE std 1003.1-1996 POSIX part 1: System Application Programming Interface*, 1996

[3] S. Karlsson and Mats Brorsson, A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing, to appear in *proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), November 2004*

[4] S. Karlsson, *Balder – An OpenMP run-time library for clusters of SMPs*, Technical report, TRITA-IMIT/LECS R 04:01, ISSN 1651-4661, ISRN KTH/IMIT/LECS/R-04/01--SE, Department of Microelectronics and Information Technology, Royal Institute of Technology, KTH, August 2004

[5] S. Karlsson, *OdinMP homepage*, http://www.odinmp.com, retrieved on August 8th 2004

[6] OpenMP consortium, *OpenMP C and C++ Application Program Interface, Version 2.0*, 2002

OpenMP on Distributed Memory Computers with FDSM Distributed Shared Memory System

Hiroya Matsuba Yutaka Ishikawa
Graduate School of Information Science and Technology
The University of Tokyo
{matsuba, ishikawa}@is.s.u-tokyo.ac.jp

Abstract

FDSM, a software distributed shared memory system, has been designed and implemented to run OpenMP programs on distributed memory computers. FDSM inspects the access pattern of an application at the first iteration of a loop and figures out the communication pattern. FDSM is implemented on both IA-32 and IA-64 architectures and evaluated using the CG benchmark in the NAS parallel benchmarks. The result shows FDSM performs better than another SDSM system called SCASH.

1 Introduction

OpenMP programs run on the distributed memory parallel computers by using the software distributed shared memory(SDSM) system[5]. It is known that the performance of SDSM is limited because of the heavy overhead to maintain the coherence of the shared region.

FDSM[6] is the SDSM system for the applications that use loops. Many scientific applications use loops and some of those access the shared region with a fixed pattern. FDSM targets such applications. FDSM obtains the access pattern to the shared region at the first iteration of a loop. By using this access pattern, the communication pattern is determined and it is used after the second iteration. If the communication pattern is the same in every iteration, the overhead to determine the communication pattern is reduced.

FDSM assumes the Omni/OpenMP compiler environment. A code generated by the Omni/OpenMP compiler runs on FDSM.

We have proposed the method to inspect the access pattern on the IA-32 architecture in [6]. This paper proposes the new method to get the access pattern on the IA-64 architecture using its debug facilities.

FDSM is evaluated using the CG benchmark program in the NAS Parallel Benchmarks on the IA-32 and IA-64 architectures. The result shows FDSM is faster than another software distributed shared memory system called SCASH on the IA-32 architecture. The result also shows FDSM performs almost same as the MPI version on the IA-64 architecture.

2 Target Applications

FDSM requires the applications to meet the following three conditions. The first condition is that the applications on FDSM must access the shared memory region with the fixed pattern. The second condition is that applications must explicitly declare the access to the shared memory region before they actually access the shared region if each process accesses the different variables by this access. The third condition is that applications must call the barrier operations when a memory barrier is required.

FDSM is designed to run the scientific applications on it. Scientific applications tend to use loops especially if they adopt iterative methods. Some of such applications access the shared memory region with the fixed pattern in each iteration of the loop, which meets the first condition.

It is the programmers' role to guarantee that the application meets the first condition. The second and third conditions are automatically satisfied by the Omni/OpenMP compiler as shown in the next section.

3 OpenMP programs on FDSM

3.1 Compilation

FDSM uses the Omni/OpenMP compiler to translate an OpenMP program to a normal C language source program. Omni/OpenMP compiler generates a source code for the SCASH[3] distributed shared memory system. When

```
/* The access pattern is fixed in the outer loop */
for (i = 0; i < IMAX; i++) {
  #pragma omp for
  for (j = 1; j <= JMAX; j++){
    z[j] = z[j] + alpha * p[j];
    r[j] = r[j] - alpha * q[j];
  }
}
```

Figure 1. OpenMP source

```
/* The access pattern is fixed in the outer loop */
for (i = 0; i < IMAX; i++) {
  {
    auto int j_41;  auto int j_42;  auto int j_43;
    (j_41)=(1);
    (j_42)=(JMAX+(1));
    (j_43)=(1);
    _ompc_default_sched(&j_41,&j_42,&j_43);
    for((j)=(j_41);(j)<(j_42);(j)+=(j_43)){
        (*((z)+(j)))=((*((z)+(j)))+((*__G__L_alpha_4)*(*((p)+(j)))));
        (*((r)+(j)))=((*((r)+(j)))-((*__G__L_alpha_4)*(*((q)+(j)))));
    }
  }
  _ompc_barrier();
} /* end i loop */
```

Figure 2. Converted OpenMP program

Omni converts an OpenMP program for SCASH, it relocates global variables to the shared region and parallelizes loops as indicated by the OpenMP directives. Figures 1 and 2 show an example of the source translation by the Omni/OpenMP compiler. Figure 1 is the original source program and Figure 2 is the translated code for SCASH.

As shown in Figure 2, the code generated by the Omni/OpenMP compiler does not directly call the SCASH API functions. For example, when the barrier operation is required, it calls **_ompc_barrier** instead of the SCASH API function. This means that the program generated by the Omni/OpenMP compiler runs on FDSM if the FDSM dedicated run-time library is linked. No modification of the Omni/OpenMP compiler is required.

Let us confirm that the source code shown in Figure 2 satisfies the conditions described in Section 2. Shared memory is accessed in the inner loop and the accessed region does not depend on the loop variable of the outer loop, i.e. the variable **i**. This means the access pattern is fixed in each iteration of the outer loop. So, the first condition is met. The second condition is satisfied because the function **_ompc_default_sched** indicates the beginning of the shared access. Although the original role of this function is to schedule the execution of the loop to share the work among the processes, this function may be used to tell the FDSM run-time library that the shared access begins. The third condition is satisfied because the function **_ompc_barrier** is called to perform the barrier operation.

3.2 Overview of FDSM

This section describes how FDSM provides a virtual shared memory system. The basic idea of FDSM is shown in Figure 3. In the first iteration of a loop, FDSM obtains the access pattern of the applications and figures out the communication pattern as shown in Figure 3 (a). We call this execution the *inspection mode*.

In the rest of iterations, the communication is performed directly from the writers to the readers using the communication pattern obtained in the *inspection mode* as shown in Figure 3 (b). We call this type of execution the *fast execution mode*.

Let us define a term *access block* for the later discussion. The *access block* means a part of program where the each

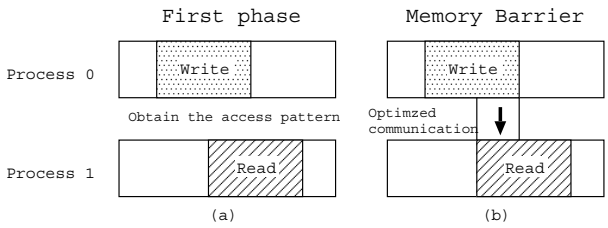


Figure 3. The idea of FDSM

process accesses to the assigned area of the shared region, i.e. the work is shared among the processes. In the example in Figure 2, the part between the **_ompc_default_sched** and **_ompc_barrier** forms a *access block*.

3.3 Inspection Mode

In the *inspection mode*, FDSM inspects the access pattern to the shared region. At the same time, FDSM provides the shared memory space with a home-based mechanism. In this mode, the application not only inspects the access pattern but also performs the normal calculations.

3.3.1 Access Pattern Inspection

In order to obtain the accurate communication pattern, the access pattern of the write access must be obtained with the 1-byte granularity. So, FDSM obtains the access pattern with this granularity. In order to reduce the read page faults, FDSM obtains the read access pattern with the page-size granularity. Unlike the write access, the 1-byte granularity is not necessity because even if FDSM judges a shared variable to be read but it is not actually read, this does not affect the correctness of the execution. We have already proposed the method to employ the page protection mechanism to get the access pattern in [6]. The overview of this method is as follows.

1. All the pages are made write protected.
2. When the application tends to write to the shared region, the page fault exception occurs.
3. The page fault handler records the accessed address and emulates the the instruction causing the page fault.
4. The execution is continued from the next instruction that caused the page fault.

Access Pattern Inspection on IA-64

On the IA-64 architecture, the access pattern may be obtained with more sophisticated way by using the facilities to debug programs. Some modern architectures including IA64 provide provide **data break points**. This break point is set by the debugger on a memory address. The difference between a normal break point and a data break point is that a data break point is set on the memory address which contains data, not an instruction. When the application program loads from or stores to a memory region specified by the data break point, the debug exception occurs.

The IA-64 architecture provides the advanced data break point. On this architecture, the data break point may be not only a single address but also the continuous memory region. Using this debugging facility, FDSM obtains the access pattern in the following way:

1. The whole shared region is set to the data break point.
2. When the application accesses the shared region, the debug exception occurs.
3. The exception handler records the accessed address.

As shown above, FDSM needs not emulate the instruction that caused the debug exception. on the IA-64 architecture, while FDSM must emulate it on the IA-32 architecture. This difference comes from the difference of the kind of the exception. As for the page fault, the exception handler must remove the cause of the exception. After the execution of the exception handler finished, the instruction which caused the exception is re-executed and it should succeed. So, the page fault handler must remove the write protection of the pages on FDSM. But if FDSM does so, the future access to this page will not cause the exception and the access pattern cannot be obtained with the 1-byte granularity. This is because FDSM must emulate the instruction.

On the other hand, the debug facility provides the way to recover from the debug exceptions. In the debug exception rule, the instruction at the exception point is executed without more exception after returning from the debug exception handler. Because of this rule, FDSM needs not emulate the instruction.

3.3.2 Shared Memory Space in Inspection Mode

While FDSM inspects the access pattern of the application, it provides the shared memory region at the same time. This enables the application to perform the normal computation in the *inspection mode*.

Because all the access blocks are executed with the *inspection mode* when it is executed at the first time, FDSM knows the access pattern of all the previous executed access blocks. FDSM also records the time when a access block is executed. FDSM is able to compute the latest writer node of any shared area at any point of the execution. Figure 4 shows an example of this computation.

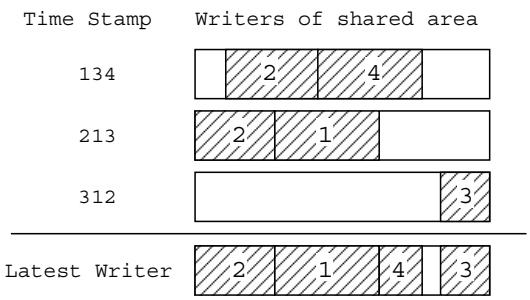


Figure 4. Calculation of the latest writer

At the beginning of the *inspection mode*, FDSM calculates the latest writer of each shared area and notifies the writer to send the latest value to the home node. Each shared area has one home node and it may be arbitrary chosen. In the current implementation of FDSM, the home node is determined using the round-robin fashion.

During the execution of an access block, the exceptions are raised to inspect the access pattern. While the exception handler records the accessed address, it copies the latest value from the home node. So, the user process correctly read the shared value.

3.4 Fast Execution Mode

In the *fast execution mode*, all the communication required to keep the coherence is performed at the beginning of each access block. FDSM knows the access pattern of all the previous executed access blocks and it also knows the access pattern of the next access block. So, FDSM is able to find out the required communication pattern so that the coming shared read may return the correct value. The communication pattern is decided as follows:

1. The latest writer of each shared area is computed.

2. The latest writer and the area which will be read in the next access block is compared. Communication is required in the area where its writer and reader is different.

Once the communication pattern is determined, this pattern may be used later when the same access block is executed again.

In the *fast execution mode*, no exception occurs during the execution of the user application. The overhead to maintain the memory consistency is only the time for communication.

3.5 Shared Access Outside of the Access Block

Applications may access the shared region outside of an access block. We define such a section as *semi access block*, which corresponds to a part of program outside of "omp for". In the *semi access block*, as for the write access, all the processes concurrently write to the same address and it should be the same value. So, the memory consistency is automatically kept. As for the read access, all the previous written value must be read. So, FDSM obtains the read access pattern in the *inspection mode* and maintains the memory consistency in the same way as the normal *access block*.

4 Implementation Issues

4.1 IA-32

The IA-32 version of FDSM is implemented on the Linux kernel 2.4.21. This version of FDSM uses the page fault exception to obtain the access pattern in the *inspection mode*. Although the access pattern inspector may be implemented as the signal handler in the user space, we have implemented it in the kernel to reduce the overhead of the context switch. So, the page fault handler in the Linux kernel is modified. The instruction emulator is also implemented in the kernel.

The user level library of FDSM has two roles. One is to provide the application programming interfaces shown in Table 1. Another is to figure out the communication patterns in the *fast execution mode*.

FDSM runs on the SCore Cluster System Software[8]. PM/Myrinet[9] is used as the communication library.

Table 1. Provided API's

fdsm_init	Initializes FDSM
fdsm_finalize	Finalizes FDSM
fdsm_alloc	Allocates the shared region
fdsm_free	Frees the shared region
fdsm_before_loop	Tells FDSM the shared area will be accessed
fdsm_after_loop	Performs Barrier
fdsm_lock	Acquires a lock
fdsm_unlock	Releases a lock
fdsm_reduce	Performs a reduction operation

4.2 IA-64

The IA-64 version of FDSM is also implemented on Linux 2.4.21. This version of FDSM uses the debug facility

Table 2. Specification of the PC Cluster

	IA-32	IA-64
# of nodes	8	4
CPU	Xeon 2.8GHz	Itanium2 1.3GHz
Cache	512KB L2	3MB L2
Chipset	Intel E7501	
Memory	2Gbtes	2GB
Network	Myrinet 1.25Gbps	Myrinet 2.5Gbps
	Lanai 4.3	Lanai 9.2
	SAN cable	SAN cable

to get the access pattern. In order to use this facility, the debug registers must be correctly accessed. An IA-64 processor has at least 4 pairs of debug registers used to set the data break points. One pair consists of address and mask registers. Let the values of address and mask registers be a and m, respectively. When the memory access to the address R is issued, if $R \,\&\, m = a$, the debug exception occurs.

FDSM sets these debug registers using the ptrace system call. The access pattern inspector runs as a parent process and the user program runs as a child process. In the *inspection mode* when the user process accesses the shared region, the debug exception is reported to the parent process. The parent process records the accessed address and continue the execution of the child process.

The run-time library of the IA-64 version is same as that of the IA-32 version. The provided API's are also same as the IA-32 version and they are shown in table 1. As the IA-32 version, The IA-64 version of FDSM runs on the SCore Cluster System Software[8]. PM/Myrinet2k64 is used as the communication library.

5 Performance Evaluations

FDSM is evaluated using the CG program in NAS Parallel Benchmarks[1]. We used the OpenMP C language version of CG and the class is B. The OpenMP program is converted to a C program by the Omni/OpenMP compiler and linked with the FDSM library. The C compiler is gcc 3.2.2 on both the IA-32 and IA-64 architectures. This program runs on the 8-node IA-32 and the 4-node IA-64 clusters. The hardware environments are shown in Table 2.

Table 3 shows the performance. Figures 5 and 6 show the speedup on the IA-32 and IA-64 architectures, respectively. In these figures, the result of SCASH[3] (only IA-32) and the MPI is also shown as a comparison. SCASH is another software distributed shared memory system which runs on the SCore system software. OpenMP programs may run on SCASH using the Omni/OpenMP compiler.

On the IA-32 architecture, in the CG benchmark running

Table 3. CG

(Mops)

	1	2	4	8
FDSM(IA32)	68.38	128.5	234.7	389.9
SCASH(IA32)	68.38	114.7	195.7	281.7
MPI(IA32)	73.91	290.3	480.1	948.2
FDSM(IA64)	72.3	146.3	265.3	
MPI(IA64)	67.5	141.9	247.1	

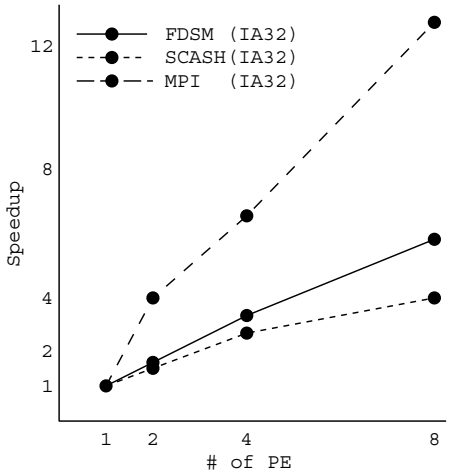


Figure 5. Speedup of the CG on IA-32

with 2, 4, and 8 nodes, the execution is 1.88, 3.43, and 5.70 times faster than one node and FDSM is faster than SCASH by 1.12, 1.20, 1.38 times, respectively. The MPI version is mush faster than the FDSM version. One reason is that FDSM has heavy overhead in the *inspection mode*. Another reason is that the MPI version of CG employs the block dimensional division to share the work among the processors while only the one-dimensional division is possible with OpenMP.

On the IA-64 architecture, the FDSM version performs almost same as MPI version. About the IA-64 version, this is the preliminary evaluation, so GCC is used as a backend compiler of the Omni/OpenMP compiler. More sophisticated compiler is required to exploit the performance of the IA-64 processor.

6 Related Work

The Inspector/Executor method[7] is similar to FDSM in that it obtains the access pattern of the applications. In this method, a special compiler divides a loop into two phases: the inspector and the executor. The access pattern of the

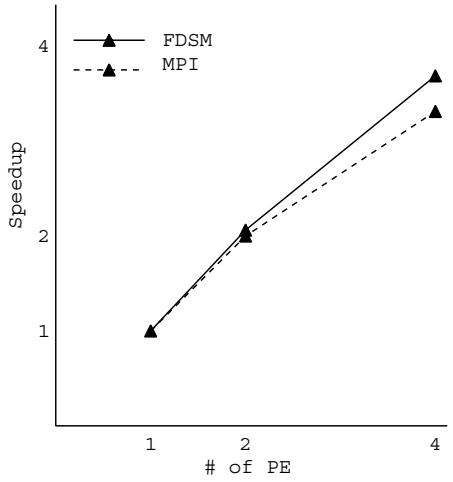


Figure 6. Speedup of the CG on IA-64

application is obtained by the inspector and the normal calculation is performed by the executor. Unlike the Inspector/Executor method, FDSM does not use the special compiler. While the inspector only gathers the access pattern of the application, the *inspection mode* of FDSM performs the normal calculations.

The producer-push method[4] is also similar to FDSM. It uses the execution history and predicts the next required communication. This method is the improvement of the twining and diffing method[2]. FDSM is different because it does not use twins nor diffs.

7 Conclusion

This paper has described the FDSM software distributed memory system. FDSM targets the application that uses loops. FDSM obtains the access pattern to the shared memory region at the first iteration of a loop. Using this access pattern, the communication is able to be optimized in the rest of the iterations.

FDSM has been implemented on the IA-32 and IA-64 architectures. This paper has described a method to implement FDSM on the IA-64 architecture. On this architecture, the access pattern of the application is obtained by using its advanced debugging facility.

For the evaluation of FDSM, the CG benchmark application has been used. The result shows FDSM performs better than SCASH by 38% on the IA-32 architecture. In the preliminary evaluation on the IA-64 architecture, FDSM performs almost same as the MPI version.

The current version of FDSM on IA-64 incurs heavy overhead because it is implemented as a user level library. FDSM on IA-64 may perform better if it is implemented in the operating system kernel.

Acknowledgment

This research is partly supported by the Grant-in-Aid of Ministry of Education, Culture, Sports, Science and Technology in Japan, No. 14208026, 2002-2004.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[3] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *Proc. of HPC Asia 2000*, pages 158–163, 2000.

[4] Sven Karlsson and Mats Brorsson. Producer-push - a protocol enhancement to page- based software distributed shared memory systems. In *ICPP 1999*, 1999.

[5] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[6] Hiroya Matsuba and Yutaka Ishikawa. OpenMP on the FDSM software distributed shared memory. In *Proceedings of the Fifth European Workshop on OpenMP (EWOMP '03)*, pages 71 – 78, September 2003.

[7] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[8] SCore. http://www.pccluster.org.

[9] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. 2000.

Session 4: OpenMP tools 1

Chair: Per Öster, KTH, Sweden

- A. J. Dorta, A. González-Escribano, C. Rodríguez, and F. de Sande, The OpenMP Source Code Repository: an Infrastructure to Contribute to the Development of OpenMP
- F. J. L. Reid and J. M. Bull, OpenMP Microbenchmarks Version 2.0

The OpenMP Source Code Repository: an Infrastructure to Contribute to the Development of OpenMP

Antonio J. Dorta, Casiano Rodríguez and Francisco de Sande
Dpto. de Estadística, I. O. y Computación
Universidad de La Laguna
38271 La Laguna, Spain
Email: {ajdorta, casiano, fsande}@ull.es

Arturo González-Escribano
Dpto. de Informática
Universidad de Valladolid
Spain
Email: arturo@infor.uva.es

Abstract

The OpenMP source code repository is based on a set of representative applications and it is supported by a web site. It is an infrastructure that we make available to the community of OpenMP users with the purpose of promoting discussions, and results interchange in the field of OpenMP research.

We want to appeal the OpenMP community to use the repository. Researchers may use it to publish the source codes developed and used for their experimental research and it is also a useful tool from the perspective of widespreading the OpenMP usage and knowledge.

The discussions originated by the use of this framework will define the real needs of the users and developers community, and they will lead the future of this work.

1 Introduction

Usually, when a research team publishes a work containing computational results, the corresponding source codes are not available to the scientific community. Upon request, authors usually provide a copy of their source codes, but even in this case, many details concerning to the experiment may still remain unrevealed.

In this work we present the OpenMP Source Code Repository (`OmpSCR`), a repository of applications parallelized using OpenMP. `OmpSCR` [1] is based on a set of OpenMP programs, but it is more than this: it is an open, non proprietary and freely accessible infrastructure oriented to share OpenMP programs with the aim of promoting discussions and results interchange in the field of OpenMP research.

Besides storing representative source codes written in OpenMP, `OmpSCR` provides some services that allow the OpenMP community to actively and dynamically discuss about aspects related to these codes. We have created `OmpSCR` with the belief that this structure will be a useful tool for the future development of OpenMP, and that setting up this framework is a need if we want to widespread the use of OpenMP.

The remaining of the paper is organized as follows: In section 2 we deepen in the motivations of this work and we outline our goals. Section 3 is dedicated to explain aspects related to the source codes currently included in the repository. In section 4 we show the most relevant characteristics of the web site that supports the repository. Some computational results obtained for different platforms using the programs in the repository are presented in section 5. Finally, section 6 offers conclusions and comments on future work.

2 `OmpSCR`: the OpenMP source code repository

It is not always easy to know the details about how a computational experiment has been accomplished. Details like the way in which execution times have been measured, the points in the code where they have been measured, the arrangement of the input data for the application or the size of these data are some of the factors to be considered.

Lets consider [2] as an example. Among the computational results presented in this work the authors show the speedup obtained when sorting a 10^6 elements vector using the *Quicksort* algorithm. Reading the information provided in the paper about the experimental framework the reader can know the compiler, the platform, the number of processors involved or the speedups obtained. Nevertheless there are fundamental details that do not result clear. All these details are not specified when writing a scientific paper due to space limitations. However, it is crucial to know them to understand a work and to establish its relevance. A similar situation occurs in many other works where computational experiments are involved.

We believe that a first step to know the way in which experiments are performed is to have some mechanism that allows access to the source code. Furthermore, in order to compare results it is very advantageous to have source codes with a similar programming style, well documented, with similar execution interfaces and with an agreed methodology for performance measurement.

This is the main idea behind the creation of OmpSCR: to provide the OpenMP users with an infrastructure that allow both to evaluate the performance of OpenMP codes and to compare it across different platforms and compilers. It should be a framework that simplifies, as far as possible, the process of massive experimentation and results processing. It will rapidly allow to choose representative applications. The results obtained will be easily comparable with those coming from other groups.

With the recent arrival of OpenMP as *de facto* standard for shared memory parallel systems programming, the need of applications that allow to measure the performance has become clear. If recent works in this field are studied, we find that two different approaches have been used: initially the researchers used a limited set of applications to test their implementations [2]. In the recent time, this lack has been partially covered by the benchmarks developers providing OpenMP versions for existing benchmarks or developing specific new OpenMP benchmarks [3], [4], [5].

The most popular benchmarks in this field (SPEC and NAS) are highly oriented to a pure performance analysis and mostly they pay attention to numerical applications and to simple parallelization of loops. It is difficult to find in these benchmarks, for example, applications that use multilevel parallelism [6], a characteristic of the OpenMP standard widely discussed in OpenMP conferences. This is the motivation for many researchers that still continue using specific applications that make evident situations usually not considered in the classic benchmarks.

One of the goals of the OmpSCR is to standardize a set of applications to be an alternative to these classic benchmarks. The characteristics that the OmpSCR applications should fulfill are:

- They should be coded using ANSI C/C++ or standard Fortran90/95.
- Both kernel applications and complete applications can be considered to be included in the repository.
- In any case they should be programs reasonably simple (up to 5000 code lines). This will ease their use by non-expert OpenMP users.
- Applications will be representative of the real world, or applications that make evident situations that require special treatment when programmed using OpenMP.
- They should be codes extensively tested and validated in a wide variety of systems.
- The code must be written according to quality standards.
- The applications must be well documented.
- It is a desirable property that the application be self-checking (containing code to test the correctness of its results).
- The programs must be freely distributed, and not affected by copyrights.

The approach that we plan to follow in the development of the OmpSCR is collaborative and incremental. We hope that different research groups contribute their own codes to be added to the repository, as far as we know that many groups use codes compliant with the above characteristics. We think that the repository must be open, not limiting the number of applications available on it. Any contribution can be incorporated if it fits the former requisites.

Initially we do not plan the creation of an OpenMP benchmark, but we do not discard this future evolution.

3 Applications in the OmpSCR

The number of applications currently available in the OmpSCR is 10 but the OmpSCR team is actively working to add new applications. The number of programs available will clearly increase depending on the interest raised on the OpenMP users and developers community.

Although most of the applications in the repository are coded using C, the OmpSCR is not oriented to a specific programming language. One of the aims of OmpSCR is to make available the same code written in different languages to check the behavior of the corresponding compilers.

The kind of applications included in OmpSCR goes from very simple examples of OpenMP parallelizations to complex codes with several parallel loops. Some of the programs constitute examples for different parallelizations that are not usual or that can not be easily implemented using OpenMP.

It is also our goal to include applications with specific interest for OpenMP compiler designers. For instance, some applications requiring multilevel parallelism have been included. Also there is an example of parallel graph search using a master-slave paradigm. OmpSCR is also thought as a repository of examples useful for learning how to properly use OpenMP. Thus, some of the programs are provided in several versions using different parallelization strategies or primitives.

For simple codes showing different ways of using OpenMP primitives it is even possible to include in the

repository bad-parallelized examples. For instance, we have included some examples of loops with carried dependences. The bad solutions degenerate in sequential codes or produce inconsistent results. These implementations are clearly marked as wrong in the distribution.

OmpSCR is distributed as a whole in a *tarball* archive. It contains a directory structure which may be installed in a local account. The distribution contains simple installation instructions, a guide for new applications developers and a FAQ file.

Inside the distribution, the directory `applications` contains specific subdirectories for each program or set of versions. In the download section of the OmpSCR web site, the applications are also available individually to be installed under the `applications` directory.

The only platform dependent information needed to compile the OmpSCR applications is the name of your OpenMP compilers front-ends and the appropriate command-line options for them. The final users must configure these details editing a configuration file or using an interactive script. Templates with usual options for common compilers and development platforms are also provided in the distribution.

To avoid vendor-specific `make` utility details the applications building is controlled with `GNU make`. In each application directory, its developer provides a very simple `GNUmakefile`. A template for this file is also provided in the distribution to be adapted by the new application developers. Usually, they only need to include the target binary name in a variable. At compile time, this standard `GNUmakefile` includes a common file with all the implicit rules needed to build C or Fortran applications, using the compiler details provided during the user's configuration. Thus, an application may be rebuild alone from its own directory, or with the whole distribution.

The programs written using each programming language are totally independent. If there is only one compiler definition, the compilation rules simply ignore the source code files for which there is no compiler definition. The binaries for the applications are stored in the `bin` directory of the distribution and command lines used to compile each application are also stored in a `log` directory for future reference.

All the applications in OmpSCR present a common style. All use the functionalities provided by a common module included in the distribution (stored in the `common` directory). This common toolkit-module unifies aspects related to the applications programmer interface. The module eases the automation of experimentation and results collection. It implements APIs for: command line arguments processing, timers definition, time measurements and execution reports generation. This module has bindings for C and Fortran90/95 languages, presenting a very similar API.

All the applications currently included in the OmpSCR distribution contains the GNU general public license. Although the OmpSCR team encourages the use of an open software license, the original author may protect her code as she wants, as far as she grants: free distribution of the source code in OmpSCR, free use of the generated program and free publication of performance results on any platform. Our purpose is to open the possibility for industry programmers to show up their developments and include real or even commercial applications in the repository.

The applications currently included in the repository are:

- Computing π
- Mandelbrot set area
- Molecular dynamic
- Quicksort
- Divide and conquer fast Fourier transform
- Bailey's "6-step" FFT
- Loops with dependences
- Cellular automaton
- LU decomposition
- Graph search

All of them are very well known applications. We have simply selected those programs that we have used in the past for our experiments. We have been also guided by the intention of including some diversity in the codes: there are sinthetic codes (Loops with dependences) and real-life ones, there are simple codes (computing π) and more complex examples; there are numerical (LU decomposition) and non-numerical codes.

4 The OmpSCR infrastructure

The infrastructure of the OmpSCR web site [1] is based on PostNuke [7]. PostNuke is a portable multilingual open software content management system (CMS) written in PHP. It is specifically oriented to user communities in which users actively post contents using web browsers. We have chosen this type of tool to support OmpSCR because it simplifies the administration of such a web site. The CMS allows administrators to dynamically work with a structured environment, manipulating any kind of contents: articles, news, FAQs, file downloads, etc. With respect to a version control system, a CMS has the advantage of additional functionalities like discussion forums or interest links.

To join the community, a new user must register herself on the web site. This allows her to actively participate, posting contents, asking questions, or joining the discussion forums. Moreover, a registered user may configure her account, adapting her web site interface. Contributions from registered users must be reviewed and approved by the administrators before they are released.

We now quickly describe the most relevant modules of the web site, focusing on their practical use. However, the best information source about the web site contents and use can be found inside itself. The user will find a complete FAQ file with answers to most of the usual questions.

The *downloads* section is the core of `OmpSCR`. It allows to download the repository and other related information. Registered users may also apply to add new files to be *download* section filling up a simple form. This section has the following subsections: *Documentation*, *OmpSCR distribution*, *Publications*, and *Results*. The *OmpSCR* subsection contains the repository directory structure and applications, which may be downloaded separately. Results subsection includes performance results for the different applications measured for specific platforms. These results are shown as plots as well as raw numerical data. More results may be also supplied by users.

A registered user may also post other information items to the *OmpSCR* web site: news items, reviews, comments, or hyperlinks to interesting web sites. News are the easiest way to supply new information of any nature to the *OmpSCR* community. Usually the *news* section will be used to discuss new applications added to the repository, computational results, call for papers for related conferences, etc. Reviews and discussion forums are other alternatives to actively participate in *OmpSCR*.

5 Computational results

In this section we present some performance results obtained for some of the *OmpSCR* applications. We focus on the kind of discussions that such results may motivate, instead that on the values themselves.

Fig. 1 shows the results obtained when executing the *Mandelbrot* and *Molecular dynamics* applications on four different platforms. The `Intel` label corresponds to a shared-memory Intel-Xeon multiprocessor PC, with four 1.4 GHz processors, and a 400 MHz system-bus. The `IBM` tag represents an RS-6000 IBM machine with 375 Mhz Power3 processors with 64Gb of memory. `HP` corresponds to a Compaq HPC 320 with 1 GHz Alpha EV68 processors and 80Gb of memory. The `SGI` plot corresponds to a SGI Origin 3000 with 600 MHz MIPS R14000 processors, and 160Gb of memory. In all the platforms we have used the native OpenMP compiler. In the case of the `Intel` and `IBM` platforms we had exclusive use of the resources.

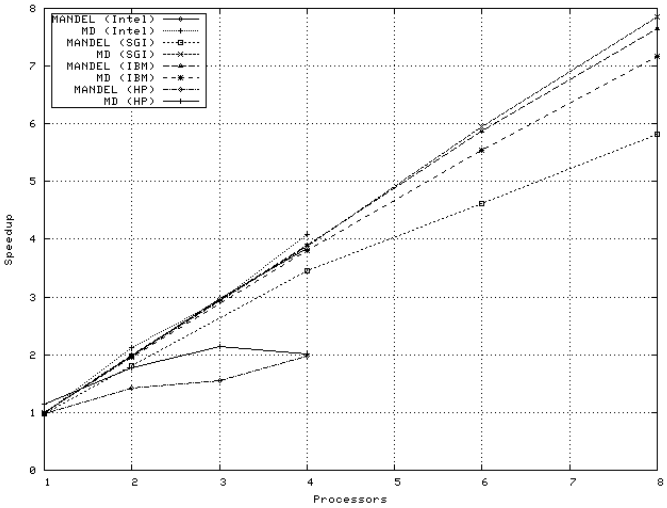


Figure 1. Results for Mandelbrot and MD applications

Both applications, Mandelbrot and Molecular dynamics, show a similar behavior for all the platforms: almost linear scalability, as expected. However, in the case of the Compaq platform, the non-exclusive use of the CPUs produces a negative impact on measures.

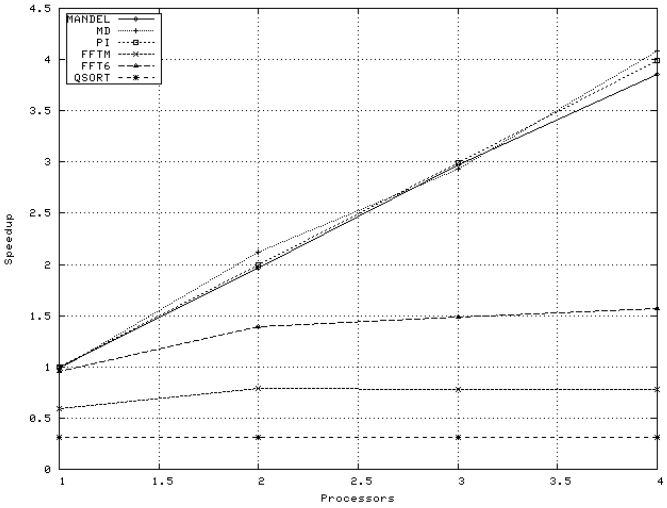


Figure 2. Results in the Intel based PC

On the other hand, Fig. 2 compares the speedups obtained on the same platform (`Intel`) for different applications. They are compiled using the Intel C++ 7.0 compiler for Linux. Applications considered are:

- *Mandelbrot* (`MANTEL`) with an input-data of 16k.
- *Molecular dynamics* (`MD`) with an input-data of 8k par-

ticles and 10 simulation steps.

- *Pi* (`PI`), computed with a precision of 10^{-8}.
- *FFT* (`FFT`), with an input signal of 2M complex data-items.
- *FFT-Bailey* (`FFT6`), with an input signal of 1M complex data-items and 10 iterations.
- *Quicksort* (`QSORT`), sorting 4M integer data-items.

The first three applications show an almost linear scalability behavior. However, `FFT-Bailey` show a logarithmic speedup, and the last two applications sublinear speedups. This effect appears because the compiler is not exploiting the multilevel parallelism present in these applications.

6 Conclusions anf future work

In this paper we have introduced `OmpSCR`, an infrastructure which purpose is to be a forum for the OpenMP users and developers community. It allows:

- To show and to evaluate different parallelization schemes for a given application.
- To evaluate and publish performance results of OpenMP codes on different platforms, using different languages or compilers.
- To provide free access to source codes and results published on the repository.
- Researches may use it to publish the source codes developed and used for their experimental research.
- It may also allow the OpenMP community to discuss about new parallel constructs and mechanisms to be included in the language, showing the effect of their proposals on real codes.
- Finally, it will be used as a resource to help in the OpenMP learning, presenting real implementation examples of the language features.

For these reasons, we want to appeal the OpenMP community to use `OmpSCR`, and we offer our support to adapt those relevant applications whose developers are interested on including them in the repository.

We have presented the current maturity-level of `OmpSCR`, a project in active development. We want to remark some of the tasks in deployment stage:

- Popularization of `OmpSCR` among the OpenMP users and developers community.
- Finding and including new applications in the repository.
- To appeal to the active working groups on this field to contribute with their own codes.
- Conducting computational experiments on different platforms, and publishing the results in the web site.

Although `OmpSCR` is not proposed as an OpenMP benchmark on its own, we do not exclude the possibility of a further development in this direction. The acceptance and contribution of the OpenMP community will determine the long term scope of `OmpSCR`. The development of a complete benchmark implies some tasks to do:

- All the benchmark applications should contain code for automatic correctness check.
- Applications should print at the end of their execution a common and complete report of performance.
- Appropriate input data characteristics and sizes must be clearly stated for each application.
- A protocol for the communication of the benchmark results should be devised.

These tasks do not present special difficulties. However, we have preferred to introduce `OmpSCR` as a repository instead of a benchmark. The discussions originated by the use of this infrastructure will define the real needs of the users and developers community, and they will lead our future works.

Acknowledgments

Computational results have been obtained using the installations of CEPBA, CESGA and CIEMAT. This work has been partially supported by the Canary Islands government, contract PI2003/113, and also by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, contracts TIC2002-04498-C05-05 and TIC2002-04400-C03-03).

References

[1] OmpSCR OpenMP Source Code Repository. `http://www.pcg.ull.es/ompscr/`.

[2] Xinmin Tian, Milind Girkar, Sanjiv Shah, Douglas Armstrong, Ernesto Su, and Paul Petersen. Compiler support of the workqueuing execution model for intel smp architectures. In *Proc. of the Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 47–55, Nice, Paris, Sep 2003.

[3] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[4] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *Proc. of of WOMPAT 2001, Workshop on OpenMP Applications and Tools*, volume 2104 of *LNCS*, pages 1–10, West Lafayette, IN, USA, July 2001.

[5] Mitsuhisa Sato, Kazuhiro Kusano, and Sigehisa Satoh. OpenMp benchmark using PARKBENCH. In *Proc. of the 2nd. European Workshop on OpenMP EWOMP2000*, Scotland, UK, September 2000.

[6] Antonio J. Dorta, Jesús A. González, Casiano Rodríguez, and Francisco de Sande. Towards structured parallel programming. In *Proc. Fourth European Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, Sep 2002.

[7] PostNuke. `http://www.postnuke.com/`.

OpenMP Microbenchmarks Version 2.0

Fiona J. L. Reid and J. Mark Bull
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, U.K.
f.reid@epcc.ed.ac.uk or m.bull@epcc.ed.ac.uk

Abstract

Overheads due to synchronisation, loop scheduling and array operations are an important factor in determining the performance of shared memory parallel programs. We present an updated set of benchmarks to measure these classes of overhead for the language constructs used in OpenMP. Results are presented for a Sun Fire 15K, an IBM p690+ and an SGI Altix, each with its own implementation of OpenMP. Significant differences between the implementations are observed, which suggest possible means of improving future performance.

1. Introduction

Synchronisation, loop scheduling and array operations can all be significant sources of overhead in shared memory parallel programs. In OpenMP ([4], [3]), the cost of these operations is dependent on their implementation in the OpenMP runtime library. In this paper we describe techniques for measuring the overheads associated with synchronisation directives, scheduling directives and array operations. We present results for OpenMP Fortran 90 implementations on a Sun Fire 15K, an IBM p690+ and an SGI Altix.

We present version 2.0 of the OpenMP microbenchmark codes which represent a significant update to version 1.0 released in 1999 [1] and the subsequent update described in [2]. The extended microbenchmark suite presented here allows the measurement of OpenMP 2.0 features and also includes a new benchmark which measures the overheads associated with array operations. The codes are available in C and Fortran 90 and can be downloaded from *www.epcc.ed.ac.uk/research/openmpbench*

The changes made to version 2.0 of the OpenMP microbenchmark codes are summarised below:

- A new benchmark measuring the overhead associated with various clauses when applied to arrays has been added. This benchmark measures the overhead of the `PARALLEL` directive with the `PRIVATE`, `FIRSTPRIVATE`, `COPYIN` and `REDUCTION` clauses and of the `SINGLE` directive with the `COPYPRIVATE` clause.
- Two new synchronisation benchmarks have been added to measure the overheads of the `WORKSHARE` and `PARALLEL WORKSHARE` directives.
- The Fortran codes have been re-written to use Fortran 90 syntax and free format; common blocks and include files have been replaced by modules.
- The repeat counts have been reduced where possible to speedup runtimes.
- `#ifdef` statements have been added to allow for OpenMP 1.0/2.0 compatibility
- The timing routines now use `OMP_GET_WTIME()` if OpenMP 2.0 is available. If OpenMP 2.0 is not available, `system_clock()` is used for the Fortran codes and `get_time_of_day()` is used for the C codes.

2. Methodology

To measure the overhead of OpenMP directives, the technique used is to compare the time taken for a section of code executed sequentially with the time taken for the same code executed in parallel enclosed in a given directive. A full description of this method is given in [1] and [2].

To obtain accurate results, careful choice of clock routine is necessary. Second differences of raw clock values are used to compute the overheads; the runtime for each measurement cannot be too large, since the measurements need to be repeated many times for statistical stability. Therefore, it is essential to ensure that the values returned by the clock routine are not only sufficiently accurate (typically to the nearest microsecond) but also sufficiently precise (preferably to 64-bit). All the results presented in this paper use the OpenMP runtime library routine `OMP_GET_WTIME()`

for the timing calculations. This routine returns 64-bit floating point values and is accurate to the nearest microsecond on the three systems measured.

To obtain statistically meaningful results, each overhead measurement is repeated 20 times within a single run and 20 different runs are performed. This means that each measurement is obtained from the average of 400 individual measurements. The reason for this is that a much larger variability is observed between different runs than between different measurements within the same run. The cause of this is unknown but it may be due to the physical memory locations of synchronisation variables altering from run to run. Although every effort is made to ensure we have exclusive access to a given machine/processors, this is not always achievable. Background processes (e.g. OS related) may also be contributing the variability observed between runs.

Within each run we compute the mean and standard deviation σ of all 20 measurements. Since we do not have exclusive access to the hardware platforms, we test for "clean" runs by checking for large values of σ and also for cases where there are many outliers (values more than 3σ above the mean).

3. Results

The benchmarks were run on the following systems: a Sun Fire 15K with 52 900MHz Ultrasparc III processors using Forte Developer, Fortran 95 version 7.0, an IBM p690+ with 32 1700MHz POWER4+ processors using XL Fortran for AIX, version 8.1.1 and an SGI Altix with 256 1300MHz Itanium 2 processors using the Intel Fortran Compiler for Linux, version 7.1. Both the Sun Fire 15K and IBM p690+ had OpenMP 2.0 compatible compilers installed. The SGI Altix did not have an OpenMP 2.0 compiler installed and therefore only results for OpenMP 1.0 directives are presented for it.

The scheduling and array benchmarks were run using 8 processors. The synchronisation benchmarks and the array benchmark for a fixed array size were run for various numbers of processors from one up to the maximum possible for the particular machine. The maximum number of processors available to a shared memory code were respectively 48, 32 and 62 for the Sun Fire 15K, IBM p690+ and SGI Altix.

Figures 1 to 3 show the measured overheads against the number of processors for the implied barrier directives on the three systems. Measurements for the `WORKSHARE` and `PARALLEL WORKSHARE` directives could not be obtained for the SGI Altix (Figure 3) as they are OpenMP 2.0 specific features.

Figures 4 to 6 show the overheads of the various clauses with array arguments against array size on the three systems. Measurements for the `COPYPRIVATE` and `REDUCTION` clauses could not be obtained on the SGI Altix as they are both OpenMP 2.0 features.

Figures 7 to 9 show the measured overheads against chunk size for the different loop schedules on the three systems.

4. Analysis

Figure 1 shows that the `BARRIER`, `DO`, `SINGLE` and `WORKSHARE` directives have similar performance on the Sun Fire 15K, suggesting that nearly all the cost of the `DO`, `SINGLE` and `WORKSHARE` directives is in the implied barrier. All four directives appear to scale well with increasing numbers of processors. The `PARALLEL`, `PARALLEL DO` and `PARALLEL WORKSHARE` directives take between 2 and 9 times as long as the `BARRIER` directive, with cost increasing steadily with the number of processors. A slight increase in cost is observed at around 44 processors that may be spurious: high standard deviations were observed for these results. The addition of a `REDUCTION` clause to the `PARALLEL` directive significantly increases its cost and makes it less scalable, especially for more than eight processors.

Figure 2 shows that the `BARRIER`, `DO`, `SINGLE` and `WORKSHARE` directives behave in a similar way on the IBM p690+ as on the Sun Fire 15K, except that the number of clock cycles is up to 12 times higher on the IBM p690+. Even allowing for the difference in clock speed between the machines these differences are large. The `PARALLEL` and `PARALLEL DO` directives are more costly (approx 30% greater for 24 processors) than the `BARRIER` directive. Unlike the Sun Fire 15K, the `PARALLEL + REDUCTION` directive costs approximately the same as the `PARALLEL` directive. This means that the cost associated with the `REDUCTION` is nearly all attributed to the cost of the `PARALLEL` directive. The `PARALLEL WORKSHARE` directive on the IBM p690+ scales very poorly, and is more than double the cost of using `PARALLEL`. All the overheads on the IBM p690+ show a distinct change in gradient at 8 processors. This is not entirely unexpected as each 32 processor frame of the IBM p690+ comprises of four eight processor Multi-Chip Modules (MCM). It appears that communications within an MCM are significantly less costly than those between MCM's where the communications need to go via the bus interconnect.

On the SGI Altix (Figure 3) the `BARRIER`, `DO` and `PARALLEL DO` directives have similar performance

and cost. As with the Sun Fire 15K and IBM p690+ the PARALLEL directive is more costly (up to a factor of two) than the BARRIER directive. The main difference is that the REDUCTION and SINGLE directives have almost the same overheads associated with them. The cost of the SINGLE directive is always greatest on the SGI Altix, e.g. for 32 processors the Altix is over 10 times slower than the Sun Fire 15K and nearly twice as slow as the IBM p690+.

Figure 4 shows the overheads of the various clauses with array arguments for the Sun Fire 15K. All the overheads except PRIVATE show a steady, linear increase with array size. The cost of the PRIVATE clause is approximately the same as a PARALLEL directive and does not vary with array size. The overhead of the COPYPRIVATE clause exhibits rather erratic behaviour for small (<729) array sizes, beyond which a steady increase is observed. The reason for this erratic behaviour is unknown. The REDUCTION clause is always the most expensive and costs more than two million clock cycles for the array size of 59049.

On the IBM p690+ (Figure 5) the overhead associated with the PRIVATE clause behaves similarly to the Sun Fire 15K, except that it is 50% less expensive on the IBM p690+. The COPYPRIVATE clause is significantly cheaper on the IBM p690+ and does not show the erratic behaviour observed on the Sun Fire 15K for small array sizes. The FIRSTPRIVATE and COPYIN clauses have nearly identical costs that increase linearly with array size. A sharp change in gradient is observed at array size 19683 suggesting that we may have encountered some architecture or implementation related limit. The overhead of the REDUCTION clause increases linearly with array size until 6561 and then levels out for larger array sizes.

Figure 6 shows the corresponding results for array clauses on the SGI Altix. The PRIVATE clause shows somewhat erratic jumps at array sizes 27, 729 and 2187; again the reason for these jumps is currently unknown. The FIRSTPRIVATE and COPYIN clauses behave in much the same way as the other two systems, except that the number cycles required is less - up to 85 times faster than the IBM p690+ and up to 11 times faster than the Sun Fire 15K.

Figure 7 shows that on the Sun Fire 15K the block cyclic scheduling (STATIC,N) costs the same as a block schedule (STATIC) for chunk sizes greater than eight. For small chunk sizes (from 1 to 8) the overhead decreases rapidly with increasing chunk size toward the STATIC value. DYNAMIC scheduling is several orders of magnitude more expensive and shows a logarithmic decrease with chunk size. GUIDED scheduling is over ten times as expensive as the STATIC,N schedule.

On the IBM p690+ (Figure 8) the overhead of the block cyclic schedule does not converge to that of the block schedule until the chunk size exceeds 32. Unlike the Sun Fire 15K, the overhead of the DYNAMIC schedule converges to that of the block cyclic schedule for chunk sizes of 64 or greater. The GUIDED schedule behaves in a similar way to the Sun Fire 15K. The DYNAMIC and GUIDED schedules on the IBM p690+ are up to three times cheaper than on the Sun Fire 15K.

Figure 9 shows that on the SGI Altix, the overheads of the block cyclic schedule and block schedule are approximately the same for all chunk sizes. Curiously, and unlike the other two systems, the GUIDED schedule is always more expensive than the DYNAMIC schedule and shows no sign of converging toward the STATIC values. Comparison with the other two systems shows that the overhead of the GUIDED schedule on the SGI Altix is more than twice that of the IBM p690+ and four times that of the Sun Fire 15K. The DYNAMIC schedule behaves in a similar manner to the other two systems and is consistently smaller, particularly for small (<32) chunk sizes.

5. Conclusions

This paper has presented an updated set of benchmarks for measuring the overheads of synchronisation, loop scheduling and array operations in OpenMP 2.0. Particular emphasis has been placed on obtaining statistically meaningful measurements. The benchmarks have been run on three distinct shared memory platforms. Differences are observed both between the individual directives and between the same directive examined on each system. Some potential areas for optimisation have also been highlighted.

6. References

[1] J.M. Bull, “Measuring Synchronisation and Scheduling Overheads in OpenMP”, *Proceedings of the First European Workshop on OpenMP*, Lund, Sweden, 1999, pp. 99-105.

[2] J.M. Bull and D. O’Neill, “A Microbenchmark Suite for OpenMP 2.0”, *Proceedings of the Third European Workshop on OpenMP (EWOMP’01)*, Barcelona, Spain, 2001.

[3] OpenMP Architecture Review Board, “OpenMP C and C++ Application Program Interface”, Version 2.0, March 2002.

[4] OpenMP Architecture Review Board, “OpenMP Fortran Application Program Interface”, Version 2.0, November 2000.

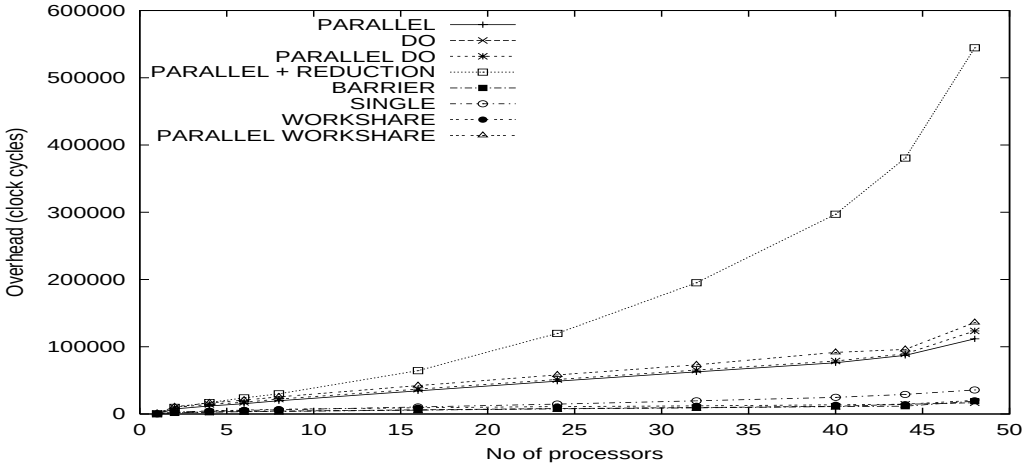


Figure 1: Synchronisation overheads on Sun Fire 15K with Forte Developer 7.0 compiler

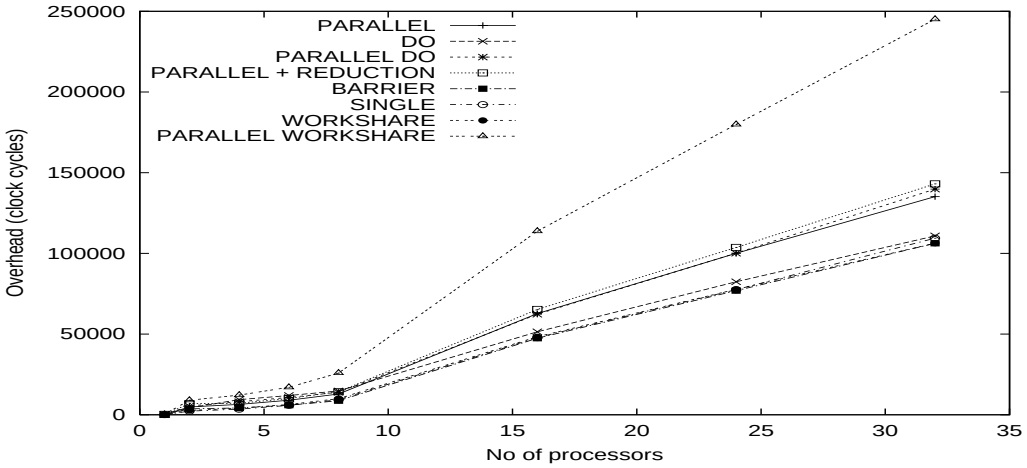


Figure 2: Synchronisation overheads on IBM p690+ with XL Fortran for AIX version 8.1.1 compiler

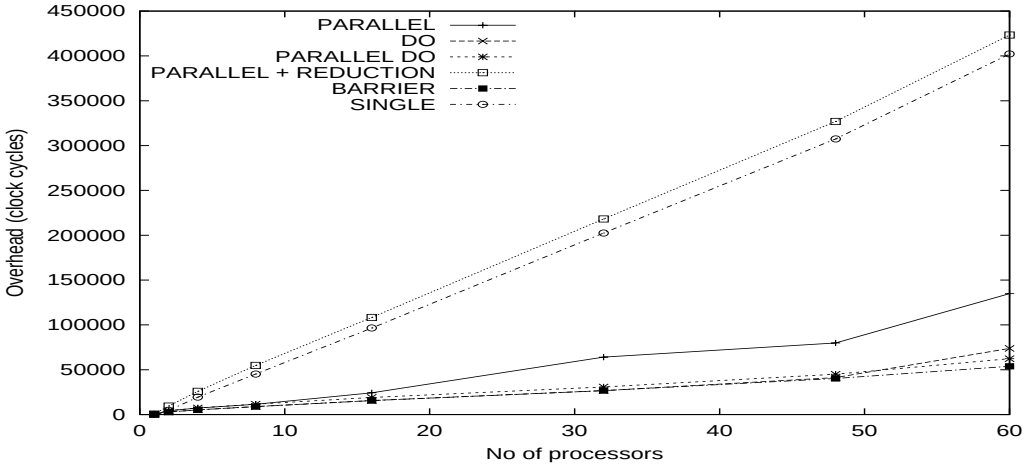


Figure 3: Synchronisation overheads on SGI Altix with Intel Fortran Compiler version 7.1

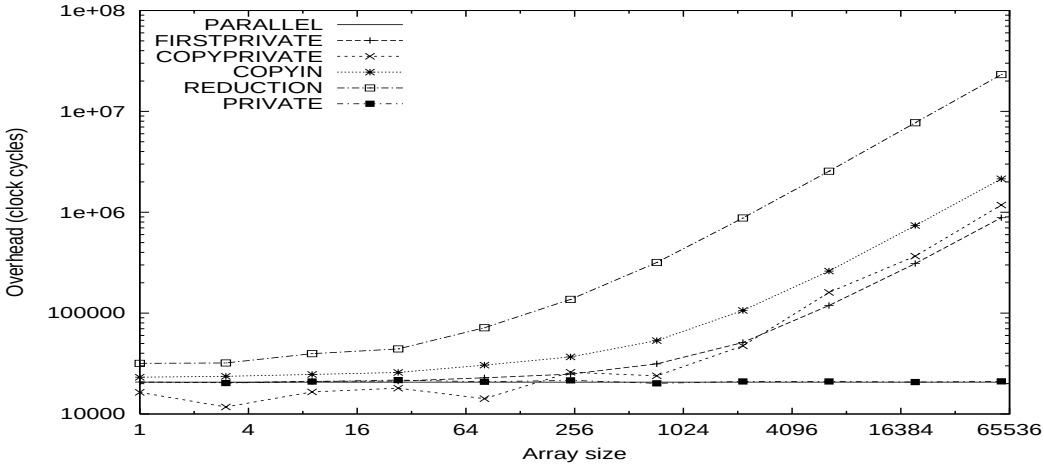


Figure 4: Array overheads on Sun Fire 15K with Forte Developer 7.0 compiler

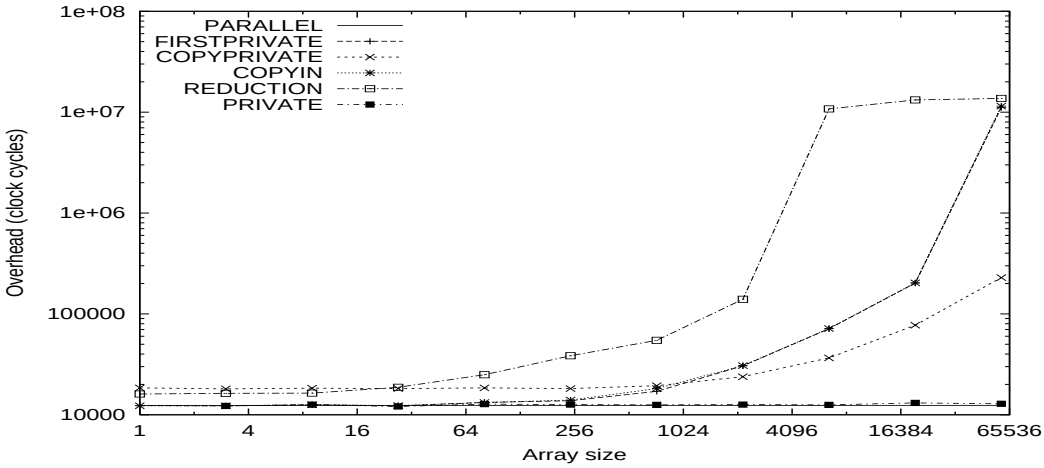


Figure 5: Array overheads on IBM p690+ with XL Fortran for AIX version 8.1.1 compiler

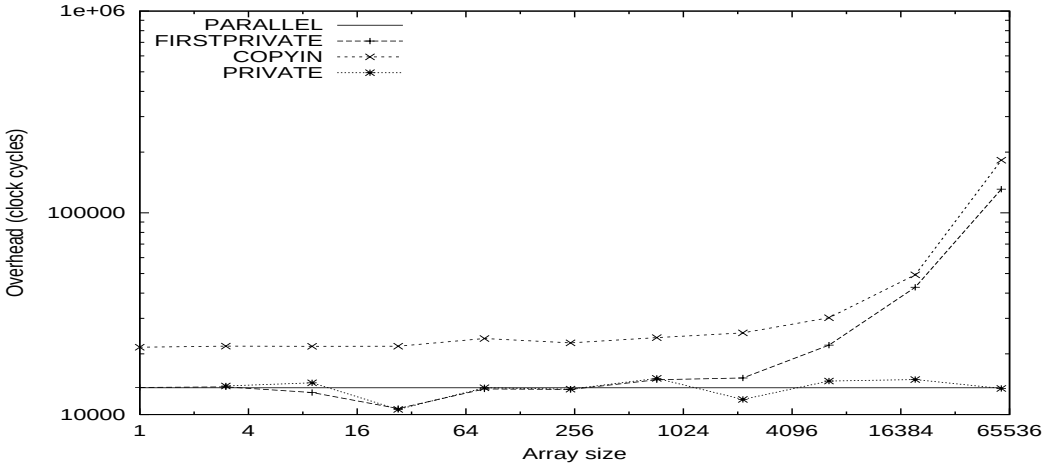


Figure 6: Array overheads on SGI Altix with Intel Fortran Compiler version 7.1

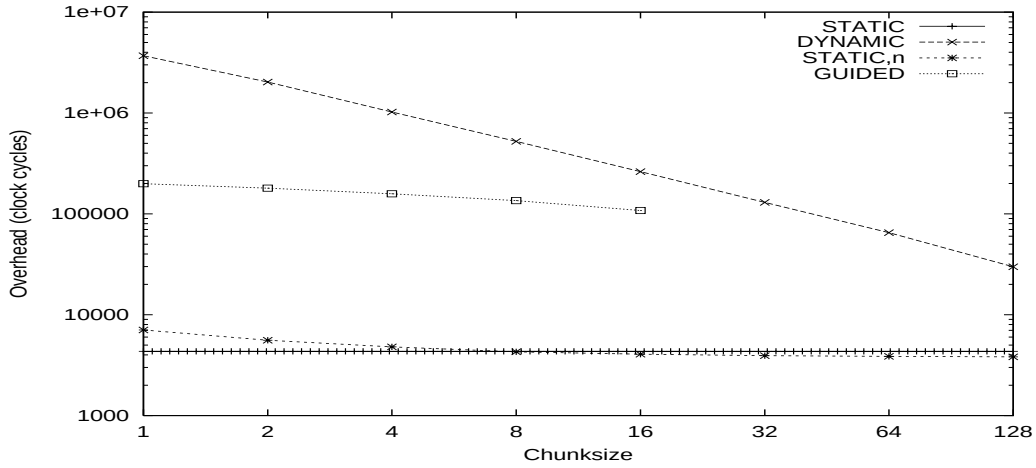


Figure 7: Scheduling overheads on Sun Fire 15K with Forte Developer 7.0 compiler

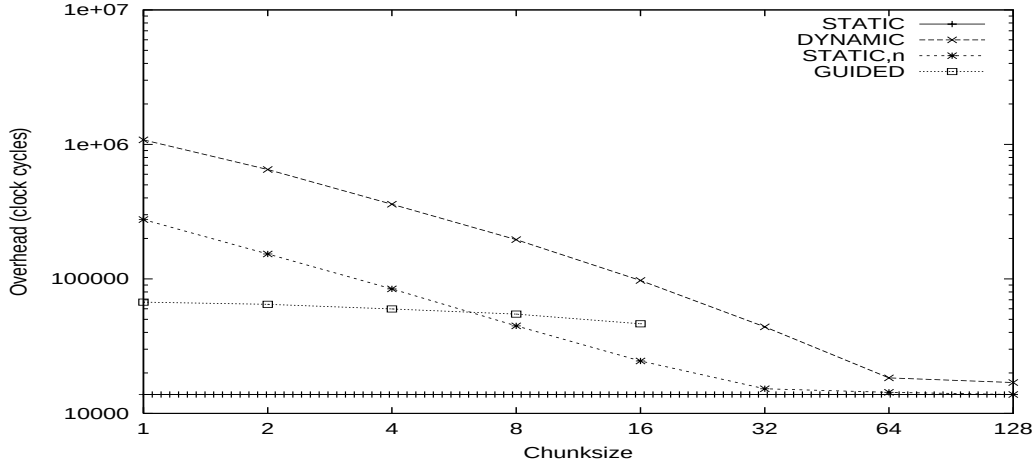


Figure 8: Scheduling overheads on IBM p690+ with XL Fortran for AIX version 8.1.1 compiler

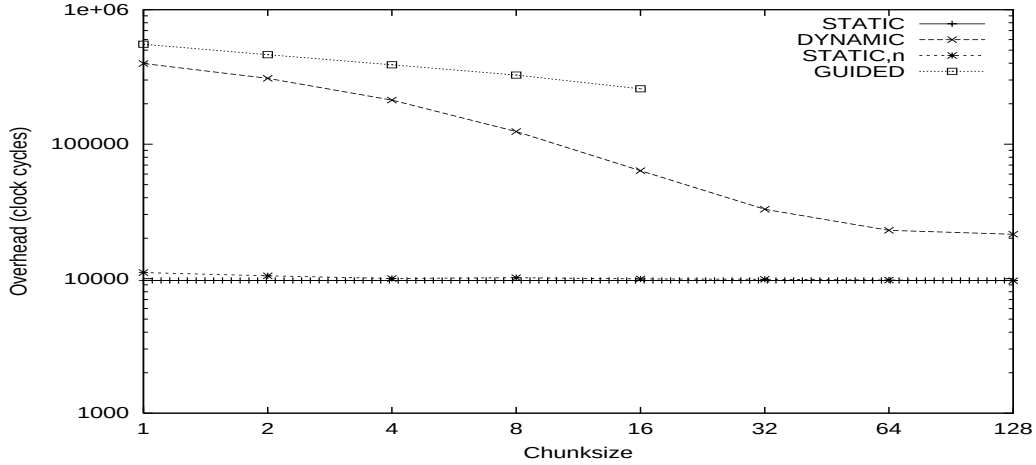


Figure 9: Scheduling overheads on SGI Altix with Intel Fortran Compiler version 7.1

Session 5: Programming models

Chair: Mark Bull, University of Edinburgh, Scotland

- P. E. Hadjidoukas and L. Amsaleg, Portable Support and Exploitation of Nested Parallelism in OpenMP
- A. Spiegel and D. an Mey, Hybrid Parallelization with Dynamic Thread Balancing on a ccNUMA system
- T. Boku, M. Sato, M. Matsubara, and D. Takahashi, OpenMPI - OpenMP like tool for easy programming in MPI

Portable Support and Exploitation of Nested Parallelism in OpenMP

Panagiotis E. Hadjidoukas Laurent Amsaleg
IRISA/INRIA, Rennes, France
{phadjido,lamsaleg}@irisa.fr

Abstract

In this paper, we present an alternative implementation of the NANOS OpenMP runtime library (NthLib) that targets portability and efficient support of multiple levels of parallelism. We have implemented the runtime libraries of available open-source OpenMP compilers on top of NthLib, reducing thus their overheads and providing them with inherent support for nested parallelism. In addition, we present an experimental implementation of the workqueuing model and the parallelization of a data clustering algorithm using OpenMP directives. The asymmetry and non-determinism of this algorithm necessitate the exploitation of its nested loop-level parallelism. The experimental results on a SMP server with four processors demonstrate our efficient OpenMP runtime support.

1. Introduction

Although nested parallelism is allowed by OpenMP [12], most OpenMP implementations do not provide the necessary support and thus nested parallel regions are always executed by a single thread. The Intel C and Fortran OpenMP compilers and the Fujitsu PRIMEPOWER Fortran compiler provide this functionality. The default Omni compiler [14] supports a limited form of nested parallelism, however Omni/ST [16], an experimental version of Omni equipped with the StackThreads/MP library, implements efficiently nested irregular parallelism. The NANOS compiler [1] also supports nested regions and OpenMP extensions for processor groups. The runtime support of the NANOS Compiler is provided by an efficient user-level threads library (NthLib), which was initially developed in the context of the NANOS project [10].

The main contributions of this paper are the following:

- An alternative implementation of the NANOS runtime library is presented. Although it shares the same architecture with the native implementation, it follows some different design decisions targeting portability and performance tricks related to unstructured multilevel parallelism.
- The runtime support of available open-source OpenMP compilation environments is implemented on top of NthLib without modifications to the compilers and the code they produce. The exploitation of user-level multithreading and some additional features introduced in NthLib reduce the OpenMP runtime overheads.
- Portable runtime support for nested parallelism is provided by a common research platform to OpenMP compilers.
- An experimental implementation of the OpenMP workqueuing model [15] on top of NthLib is also presented, as a proof of concept with respect to runtime support.
- Finally, the parallelization, using OpenMP directives, of a hierarchical data clustering algorithm that necessitates the exploitation of multilevel parallelism, is described.

The rest of this paper is organized as follows: Section 2 summarizes the design decisions and features of our implementation of NthLib. Section 3 describes the implementation of available open-source OpenMP libraries on top of NthLib. Section 4 presents our parallel data clustering algorithm. Experimental results are included in Section 5. Finally, Section 6 discusses our ongoing work.

2. Runtime Library

The NANOS runtime library (NthLib) [11] provides two different primitives to spawn parallelism, depending on the hierarchy level in which the application is working: the deepest level is generated with work descriptors, a data structure that contains pointers to a function and its arguments and results in work execution through a function call without

allowing further spawning of parallelism. Higher levels are generated using nanothreads, a different interface which provides threads with a stack. Nanothreads provide an address space for private variables and can be used at all levels. Their descriptor is an extended version of work descriptor and is always allocated in the stack of the associated user-level thread, using a single memory allocation call in case their recycling queue is empty. Nanothreads are submitted for execution in ready queues, while work descriptors are inserted in special locations in shared memory.

The alternative implementation of NthLib presented in this paper serves as a supplement of the native one. The main features that differ in our case are the significantly higher portability and the adoption of a lazy stack allocation policy. For the rest of this document the term NthLib will refer either to this alternative implementation or to some general features present at both cases, and any reference to native NthLib is mentioned explicitly.

The high portability stems from the use of the POSIX standard. Instead of using native kernel-threads, the virtual processors are system-scope POSIX threads. This feature improves NthLib's interoperability with third-party libraries and facilitates the co-existence of OpenMP and POSIX threads in the same program. In our case, user-level threads are provided to NthLib by the Underlying Threads Library (UthLib) [6], which replaces the assembly based QuickThreads package. Context-switching is implemented using setjmp-longjmp calls or ucontext operations. An underlying thread is actually a stack where the work, i.e. routine, of a nanothread descriptor is executed. Since UthLib supports thread routines with a single argument, nanothread routines are executed through an appropriate proxy routine that receives as argument the corresponding descriptor.

Another important difference is the adoption of a lazy stack allocation policy: the data structures for nanothreads and work descriptors are identical and the stack for a nanothread is allocated just before its execution. Initial motivation of this design decision was its minimal memory consumption and the easier management of parallelism on Software Distributed Shared Memory (SDSM) clusters [7, 13]. Since shared memory operations are very expensive on clusters, work descriptors are inserted in the runtime queues. In order not to harm the modularity of our implementation, we follow the same approach on shared-memory multiprocessors. Lazy stack allocation has allowed us to introduce *stack handoff*, an optimization that almost equalizes the runtime overheads between nanothreads and work descriptors. According to peer-to-peer scheduling, a finished nanothread picks the next descriptor, creates an underlying thread/stack for that descriptor and switches to that. Normally, the underlying thread is recycled from an appropriate queue. Using stack handoff, a finished nanothread simply re-initializes the underlying thread by replacing its descriptor with the new one and jumps directly to the nanothread's execution proxy routine. By allocating the descriptors from the parent's stack, the activation of the recycling mechanism for descriptors is avoided too.

A critical performance issue in multithreading environments that still lacks portability is that of synchronization: our implementation uses the spin locks and the atomic primitives available in native NthLib. These locks can be alternatively mapped to POSIX Threads mutexes or spinlocks, and they are also used to protect the ready queues. This is also the default case in native NthLib, which optionally supports lock-free queues. Finally, we have introduced a hybrid implementation of barriers between nanothreads that combines busy-waiting and blocking, according to the number of threads and the level of parallelism they belong to. This issue is further discussed in the next section.

3. OpenMP Runtime Support

OpenMP Compilers and NthLib. Most open-source research OpenMP compilers translate C or/and Fortran programs with OpenMP directives into code suitable for compilation with a native compiler linked with their OpenMP runtime library, which usually supports only a single-level of parallelism. We have managed to provide the required runtime support by implementing a corresponding software layer on top of NthLib. Although this layer could be implemented from scratch, starting from the compiler-produced code, the source code availability allowed us to apply a straightforward porting of these OpenMP runtime libraries on top of NthLib.

Initially, we applied our approach to the runtime library of the Omni OpenMP compiler. The successful integration of NthLib in this compiler motivated us to continue with the runtime libraries of the OMPi [4], ORC [3] and Intone OpenMP compilers [8]. In this paper, we focus only on the first two compilers (Omni, OMPi). The implementation of the ORC and Intone OpenMP runtime libraries was straightforward since the former is similar to that of Omni's (version 1.4), while the latter was implemented in native NthLib, in the context of the Intone project. NthLib was used to provide runtime support, single-level though, to a

modified version of the NANOS compiler and to the OdinMP C compiler.

Omni supports both OpenMP C and Fortan77 and its latest version (1.6) provides two alternative implementations of the OpenMP runtime library for shared memory multiprocessors. The default library (libompc) is based on POSIX Threads while the second one (libompst) uses the StackThreads library. The former is the most widely used due to its portability while the latter supports nested parallelism but unfortunately lacks portability and modularity. The implementation of Omni's runtime library on top of NthLib is equivalent for both versions and combines their main features: portability, user-level multithreading and nested parallelism. Omni's locks have been mapped to NthLib's locks and its thread pooling has been replaced by the virtual processors of NthLib, while generated work is represented with nanothreads. NthLib's descriptors have been extended in order to include information used in Omni's library: for instance, the descriptor of a nanothread that spawns parallelism contains all the necessary parental information the spawned threads need access to. Another issue related to user-level threading and nested parallelism was the applicability of pure busy-waiting for synchronization. It was resolved by extending the barriers between nanothreads so as to operate with an arbitrary number of threads in a team and multiple levels of parallelism. This functionality is also important when a large number of threads is spawned. Native NthLib assumes always that dynamic parallelism is enabled and thus the number of spawned threads never exceeds that of available processors.

OMPi is a recent experimental C compiler for OpenMP and provides the first publicly available implementation of the version 2.0 of the standard. For this runtime library, which makes extensive use of the POSIX Threads API, we followed exactly the same approach as for Omni. Additionally, we implemented appropriate POSIX-like barriers and condition variables in NthLib. Finally, we introduced a slight modification into the compiler in order to generate code that uses the abstract data type for NthLib's locks instead of hard-coded POSIX Threads mutex declarations.

Management of Nested Parallelism. Due to its lightweight user-level threads and the lazy stack allocation policy, NthLib can support efficiently a large number of threads and multiple levels of parallelism. Our initial approach for handling nested parallelism within the implemented OpenMP runtime libraries is a variation of the all-to-all scheme. Specifically, we distribute the descriptors spawned at the first level of parallelism to all processors, inserting them at the end of the ready queues. Moreover, we insert the descriptors of inner levels at the front of the local ready queue that belongs to the processor they were created on. In our implementation, an idle virtual processor still extracts descriptors from the front of its local ready queue but, in contrast with native NthLib, steals from the end of remote queues, favoring a virtual processor to exploit the potential data locality of inner levels of parallelism. Although Omni/ST allows an idle processor to issue random requests for work stealing, it requires the intervention of the remote virtual processor, which introduces runtime overhead and reduces portability. The latter feature is further reduced due to the dependence of Omni/ST on the StackThreads compiler, a patched version of GNU C. On the contrary, work stealing in NthLib is performed asynchronously and the access of the virtual processors to the remote queues is performed in a uniform way. Moreover, our software configuration allows the integration of any native compiler.

Workqueueing Model. The OpenMP workqueuing model is a flexible mechanism for specifying units of work that are not pre-computed at the start of the worksharing construct. To demonstrate the inherent runtime support of NthLib to this execution model, we have developed an experimental source-to-source translator that maps the `parallel taskq` and `task` pragmas to calls of the NANOS API. Although the translator is limited to programs where tasks are single function calls and no clauses are used with the directives, it is adequate for a wide variety of applications. According to [15], `parallel taskq` creates a single queue and spawns worker threads that execute the available tasks. Since NthLib maintains its own queues and workers, this pragma simply initializes the execution environment by adding one dependency to the current nanothread. For each task, the dependencies are increased by one and a nanothread descriptor is created and submitted for execution in the local ready queue. Finally, at the end of `parallel taskq`, the current nanothread suspends its execution, waiting the created tasks to finish. Our proposal for the management of nested parallelism is also applicable to nested parallel task queues.

4. Parallel Data-Clustering Algorithm

Data clustering is one of the fundamental techniques in scientific data analysis and data mining. The problem of clustering is to partition the data set into

```
 1. find_nearest neighbor(int i, int *idx, double *dist) {
 2.    min_dist = +∞, min_idx = -1;
 3.    for (j = 0; j < i; j++) {
 4.        - if (entry j has been invalidated) continue;
 5.        - if ((dist = compute_dist (i, j)) < min_dist)
 6.          { min_dist = dist; min_idx = j };
 7.    }
 8.   *idx = min_idx; *dist = min_dist;
 9. }
10.
11. update_nearest_neighbors(int pair_low, int pair_high) {
12.    for (i = pair_low; i < N; i++) {
13.       - if (entry i has been invalidated) continue;
14.       - if (entry i had neighbor pair_low or pair_high)
15.            find_nearest_neighbor (i, &nnb[i].index, &nnb[i].dist);
16.       - else if (pair_high < i)
17.            if ((dist = compute_dist(pair1, i))) < nnb[i].dist)
18.            { nnb [i].index = pair_high; nnb[i].dist = dist; }
19.    }
20. }
```

Figure 1. Pseudocode of the update phase in CURE

segments (called clusters) so that intra-cluster data are similar and inter-cluster data are dissimilar. In this section, we present a parallel implementation of CURE (Clustering Using REpresentatives) [5], a well-known and efficient hierarchical data clustering algorithm, using OpenMP. Despite its efficiency, the worst-case time complexity of CURE is $O(n^2 \log n)$, where n is the number of points to be clustered. The parallelization of the algorithm aims at enabling its direct use on very large data sets and, to the best of our knowledge, it is the first time that OpenMP has been applied to such an application.

PCURE (Parallel CURE) uses an array of records, holding information about the size, the centroid and the representative points of each cluster. According to the hierarchical algorithm, every data point is initially considered as a separate cluster with one representative, the point itself. At the initialization phase, the algorithm computes the closest cluster for each cluster. Next, it starts the clustering, merging the closest pair of clusters until only k clusters remain. When two clusters are merged, we store the information for the merged cluster in the entry of the first cluster and simply invalidate the second one. The algorithm also maintains per-cluster information about the index of the closest cluster and the minimum distance to it. To avoid duplication of this information the algorithm searches for the closest neighbor of a given cluster only in entries with a smaller index.

Figure 1 presents in pseudocode the most computation demanding routine of the clustering algorithm, which corresponds to the update of the nearest neighbors. Special features of this algorithm are its asymmetry and non-determinism, due to the way the nearest clusters are computed and the gradual decrease of valid entries (clusters). The efficiency of the parallel implementation strongly depends on the even distribution of computations to the processors. This, however, is a challenging task due to the peculiarities of the algorithm. Our experiments confirmed that the update phase cannot scale efficiently if only one level of parallelism is exploited, regardless of the scheduling policy and the chunk size used. Therefore, we have also parallelized the loop in the find_nearest_neighbor routine.

5. Experimental Results

We performed our experiments on an Intel Pentium III system with four processors at 550 MHz, with 512KB cache memory and 1GB of main memory, running Linux (2.6.6) with the Native Posix Threads Library (NPTL). As OpenMP compilers we have used Omni 1.6 (libompc) and OMPi 0.8.1 and as native compiler GNU gcc 2.3.2.

Figure 2 presents the synchronization overheads for the parallel and for OpenMP constructs, measured using the EPCC microbenchmarks [2]. These two constructs are representative of the OpenMP overheads related to threading. We observe that the overhead of parallel is lower for both compilers when the runtime support is provided by NthLib. On the other hand, when the number of threads is less or equal to the number of processors, the for construct exhibits higher overhead for native OMPi, due to the blocking POSIX barrier. However, when the number of threads increases, which can also occur in the case of nested parallelism, the overhead of native Omni's busy-wait barrier is significantly higher.

Figure 3 depicts the performance speedups over the sequential version of four benchmarks (CG, FT, LU, MG) of the NAS Parallel Benchmarks suite [9], ported

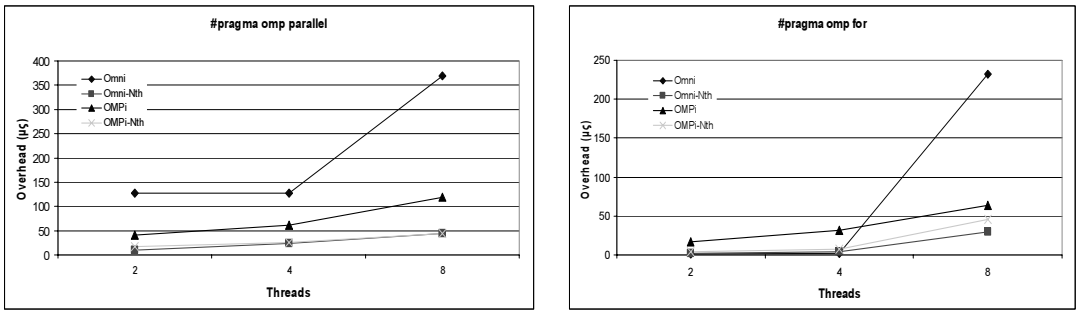


Figure 2. OpenMP overheads (parallel, for)

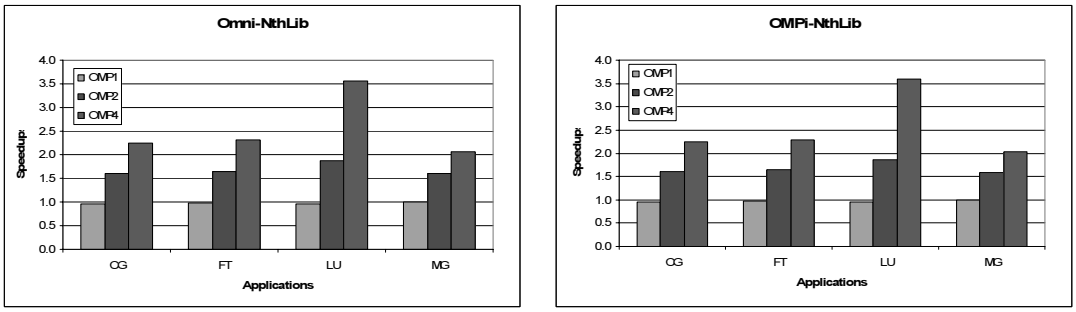


Figure 3. Performance of NAS benchmarks

to OpenMP C by the Omni group. Despite their different runtime overheads, all configurations result in very similar execution times. Therefore, results using the native runtime libraries are not included. Figure 4 shows the performance results of four benchmarks parallelized using the proposed workqueuing model. BlockLU and Raytrace come from the Splash benchmark suite [17], while FFT and Queens are two of the workqueuing benchmarks used in [15].

For our last experiment, we use PCURE as a benchmark in order to demonstrate our runtime support of nested parallelism and to provide a comparison with other OpenMP environments. For both loops of the update phase we use the static policy with chunk equal to 10. We used a small number of records (5000), of dimension 24, from a data set that contains image descriptors. Since these are organized in spherical clusters, we use a single representative per cluster. The clustering stops after 4000 steps. In Figure 5, we present the performance speedups obtained for the update phase of PCURE. The first two columns correspond to the use of the Omni and OMPi compilers with their runtime libraries built on top of NthLib. The third column (WQ) is an equivalent implementation of PCURE based on the workqueueing model. Finally the last two columns illustrate the performance speedups when the Intel (version 8.0) and Omni compilers are used. Omni uses its Posix Threads based library and the size of the thread pool (OMPC_NUM_PROC) is double the number of requested threads (x 2 levels of parallelism). We observe that the application scales only when the runtime support is provided by NthLib. The workqueuing version performs slightly better because it avoids some explicit execution barriers. We also observe that the Intel and Omni compilers provide better results when 2 threads are used. In this case, the total number of threads that execute the two nested loops is equal to the number of physical processors. For 4 threads per loop, the speedup drops mostly due to the contention of the 8 total worker threads and the increased overheads of the runtime libraries.

6. Ongoing work

We are currently working on an alternative and portable implementation of the NANOS CPU Manager. This will allow us to apply the whole runtime infrastructure of the NANOS project to other OpenMP compilation systems. Regarding data clustering, our goal is to cluster very large data sets by running PCURE on clusters of SMPs.

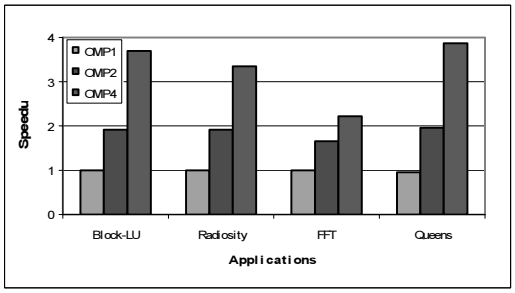


Figure 4. Performance of Workqueuing

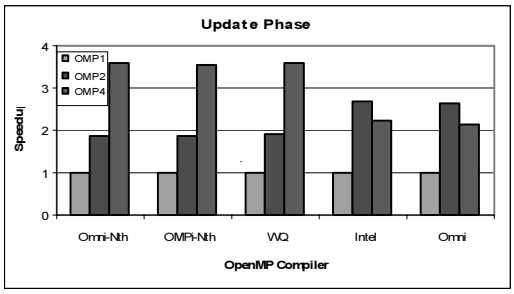


Figure 5. Performance of PCURE

Acknowledgments

Many thanks to all the partners in the NANOS and POP projects. We would also like to thank Dr. Clay Breshears for providing us the source code of the workqueuing examples.

References

[1] E. Ayguade, M. Gonzalez, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*. Special issue on OpenMP, vol. 12, no. 12. pp.1205-1218, October 2000.

[2] J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *Proc. of the European Workshop of OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.

[3] Y. Chen, J. Li, S. Wang, and D. Wang. ORC-OpenMP: An OpenMP Compiler Based on ORC. In *Proc. of the Intl. Conf. of Computer Science (ICCS)*, Krakow, Poland, June, 2004.

[4] V. V. Dimakopoulos, E. Leontiadis, and G. Tzoumas. A portable C compiler for OpenMP V2.0. In *Proc. of the European Workshop on OpenMP (EWOMP '03)*, Aachen, Germany, September 2003.

[5] S. Guha, R. Rastogi, and K. Shim. CURE: An Efficient Clustering Algorithm for Large DataBases. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 1998.

[6] P.E. Hadjidoukas. UthLib: A Portable Non-Preemptive User-Level Threads Package. Technical Report HPCLAB-TR-230304, High Performance Information Systems Laboratory, University of Patras, March 2003.

[7] P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou. OpenMP Runtime Support for Clusters of Multiprocessors. In *Proc. of the Intl. Workshop on OpenMP Applications and Tools (WOMPAT '03)*, Toronto, Canada, June 2003.

[8] INTONE: Innovative Tools for Non Experts, IST/FET project (IST-1999-20252), http://www.cepba.upc.es/intone/.

[9] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. NASA Ames Research Center, Technical Report NAS-99-011, 1999.

[10] NANOS: Effective Integration of Fine-grain Parallelism Exploitation and Multiprogramming, ESPRIT Project No. 21097, http://research.ac.upc.es/nanos/.

[11] NANOS Consortium. M2.D2: n-RTL Implementation. NANOS project deliverable, April 1998.

[12] OpenMP Architecture Review Board. OpenMP Specifications. Available at http://www.openmp.org.

[13] POP: Performance Portability of OpenMP, IST/FET project (IST-2001-33071), http://www.cepba.upc.es/pop.

[14] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.

[15] E. Su, X. Tian, M. Girkar, H. Grant, S. Shah, and P. Peterson. Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures. In *Proc. of the 4th European Workshop on OpenMP*, Rome, Italy, September 2002.

[16] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Proc. of the Fifth Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR '00)*, Rochester, NY, USA, May 2000.

[17] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations, In *Proc. of the 22th Intl. Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

Hybrid Parallelization with Dynamic Thread Balancing on a ccNUMA System

Alexander Spiegel and Dieter an Mey
Center for Computing and Communications, RWTH Aachen University, Germany
{spiegel,anmey}@rz.rwth-aachen.de

Abstract

SMP Clusters with fat nodes offer an interesting capability for large applications that employ a hybrid parallelization model: to improve load balance, the number of threads can be increased in order to speed-up busy MPI processes or decreased to slow down idle MPI processes, provided these processes reside on the same SMP node. We developed a library which performs this thread adjustment automatically during program execution. On a Sun Fire 15K ccNUMA machine the library also improves data locality by taking advantage of a low level API provided by the Solaris 9 Memory Placement Optimization feature. Experimental results demonstrate remarkable speed-ups with minimal programming effort.

1. Introduction

Hybrid parallelization using message-passing and multi-threading with OpenMP or auto-parallelization on clusters of shared-memory computers is not always profitable [1, 3], yet it offers interesting opportunities [2], particularly on SMP clusters with fat nodes. If an application employs a hybrid parallel programming model utilizing both MPI and threads (from now on we call such an application a "hybrid application" for simplicity), the number of threads can be increased in order to speed-up busy MPI processes or decreased to slow down idle MPI processes, provided these processes reside on the same SMP node.

It may not always be easy to find out the optimal distribution of threads to the MPI processes, however, and the optimal distribution may change in the course of the runtime of an application. Therefore, we developed a dynamic thread balancing (DTB) library which performs this thread adjustment automatically, requiring the user just to insert one extra MPI call in the code to trigger this feature.

The PMPI profiling interface, which is part of the MPI specification and therefore part of each compliant implementation, offers an easy and portable way of intercepting calls to the MPI library. The DTB library uses this interface to capture the user time of each MPI process and the time spent in MPI routines. This information is evaluated to adjust the number of threads for each MPI process, taking care not to oversubscribe the number of processors dedicated to the application. The major features and the implementation of the DTB library are described in section 2.

First results of experiments performed on a Sun Fire SMP cluster with the new multi-zone version of the BT code of the NAS Parallel Benchmark suite and the FLOWer Navier-Stokes solver have been described in [7]. Here in section 3 we concentrate on further experiments carried out with the FLOWer code taking the ccNUMA architecture of the Sun Fire 15K into account. Section 4 concludes with a summary and an outlook on future work.

2. The Dynamic Thread Balancing (DTB) Library

2.1. Usage of the DTB Library

The DTB library has to be linked between the user program and the MPI library. One or more calls to the newly provided routine MPI_Pcontrol have to be inserted in suitable spots of the program code in order to regularly – typically once per iteration - trigger the steering mechanism. Thus, programming effort for the user of the DTB library is minimal. The calls of the MPI_Pcontrol routine are ignored when the DTB library is not linked. The compute time and the MPI overhead between two successive calls of each MPI process are measured and evaluated to adjust the number of threads.

The DTB steering mechanism shifts threads between the MPI tasks residing on the same node with the OpenMP `omp_set_num_threads` call. If freed threads stay in a busy waiting state, they still consume

compute cycles. Therefore they have to be put to sleep completely. On Solaris we hence set the environment variable **`SUNW_MP_THR_IDLE`** to sleep. For portability reasons we would appreciate the availability of an additional parameter of the **`omp_set_num_threads`** runtime function as part of the OpenMP specifications. Of course the more MPI processes reside on one node and the more CPUs are available on a node, the better the algorithm is able to shift threads between the MPI processes and to improve load balancing.

When using multiple SMP nodes, the initial distribution of MPI processes onto the nodes is very important: If all the busiest processes are started on the same node, than the DTB algorithm has little to gain. It is thus advantageous to first execute a shorter profiling run - with the balancing mechanism of the DTB library turned off - and then use an optimized distribution for the production run.

2.2. Prediction of the Runtime of the MPI Process

In each iteration the number of threads and the compute time is stored for each MPI process. This information is then used to calculate an approximation function T(t) of the runtime of each MPI process depending on the number threads t

$$\mathbf{T(t) = T_s + T_p / t + T_o * t}$$

with

T_s = serial part of the runtime
T_p = parallel part of the runtime
T_o = parallelization overhead

by using the least square method to determine T_s; T_p and T_o.

If the program behaves smoothly, the corresponding system will have a unique solution with non-negative components. If this is not the case, for example because a loaded system impacts the program performance, we try to find a solution of a simplified approach by setting one or even two of the parameters to zero. In any case, in an overloaded system any increase of the number of threads will diminish the performance and the DTB library will therefore most likely reduce the thread number.

In order to adapt to a changing runtime behaviour, an aging factor, adjustable by an environment variable, is employed to give more recent timing measurements a higher weight.

At program start we can activate a warm-up phase in order to speed-up the initial balancing process. Thereby we quickly gather three values of T(t) for the approximation formula. If warm-up is disabled, all processes are started with only one thread and all remaining threads are put into the thread pool (see below).

2.3. The DTB Steering Algorithm

A pool of unused threads is managed by the following steering mechanism:

- Search for the MPI process taking maximum compute time (*maxwork_task)* (This process has to be accelerated.)
- Search for the MPI process with more than one thread and minimum predicted compute time with one less threads (*minwork_task*) (This process will suffer at least from losing one thread.)
- If the thread pool is empty and the predicted compute time of *minwork_task* with one less thread is less than the current and predicted compute time of *maxwork_task*, reduce the number of threads of process *minwork_task* by one and increase the thread pool (This process may loose one thread, without slowing down the whole application.)
- If the thread pool is not empty and if the predicted compute time of process *maxwork_task* with one more thread is less than the current maximum compute time, increase the number of threads of process *maxwork_task* and reduce the thread pool by one. (The busiest process really profits from an additional thread.)
- Else if the predicted compute time of process *maxwork_task* when loosing one thread is less than the current maximum compute time, decrease the number of threads of process *maxwork_task* and increase the thread pool by one. (The busiest process does not scale well. In fact, one less thread might be profitable.)

2.4. Limitations of the DTB Library

The DTB library currently is limited to the masteronly hybrid programming method (described, for example, in [3]), in which only the master thread calls the MPI library routines outside of any parallel regions. As DTB uses the PMPI interface, other MPI profiling tools like Vampirtrace can not be combined with the DTB library.

3. Experimental Results

3.1. Computing Environment

Timing experiments were performed on a Sun Fire 6800 and on a Sun Fire 15K at RWTH Aachen University. The Sun Fire 6800 is equipped with 24 UltraSPARC III Cu processors running at 900 MHz and the Sun Fire 15K contains 72 processors of the same kind. Both machines contain identical system boards with 4 CPUs plus local memory each. But whereas the Sun Fire 6800 has a rather flat memory system, the Sun Fire 15K has a ccNUMA architecture with a ratio of 1:2 concerning the latency of local versus remote memory accesses due to a different cache coherency protocol.

The production environment was Solaris 9 8/03 with Memory Placement Optimization (MPO) feature enabled and we used the Sun ONE Studio 8 compilers. The Sun HPC ClusterTools 5 package includes a fully thread-safe MPI 2 implementation. See [8] for more details about the Sun Fire Servers and MPO.

3.2. The FLOWer Navier-Stokes-Solver

In an ongoing project sponsored by the German Research Council (DFG), scientists of the Laboratory of Mechanics of RWTH Aachen University are simulating PHOENIX, a small scale prototype of the Space Hopper, a space launch vehicle designed to take off horizontally and glide back to earth after placing its cargo in orbit (see [4-6]). The corresponding Navier-Stokes Equations are solved with FLOWer, a flow solver developed at the German Aerospace Center (DLR).

The code is parallelized with the CLIC-3D communication library which encapsulates all the MPI communication. As all information exchange is block oriented, the number of blocks limits the number of MPI tasks. When less MPI processes are started than blocks are used, FLOWer distributes the blocks over the processes according to the grid sizes in an effort to equalize computational load. Underneath the coarse-grained parallelization with CLIC-3D/MPI, all but one of the compute intense loop nests can be efficiently auto-parallelized using threads.

In table 1, the runtimes of a job running on the Sun Fire 6800 with six MPI tasks, each using 2 threads, is shown in detail. It can be seen that the compute time of the various processes differs by a factor of over two.

Table 1. FLOWer on one Sun Fire 6800, 6 processes with 2 threads each, runtime of the single MPI processes in seconds

Rank-ID	Compute Time	MPI Overhead
0	258.5	253.3
1	182.8	329.0
2	210.6	301.2
3	187.5	324.3
4	186.0	325.8
5	497.6	14.2

In table 2, we show the impact of adding more threads, with or without employing the DTB library. We noticed that an increase of the number of MPI tasks not necessarily improves performance, which probably has to do with the fact that a greater number of MPI tasks leads to a greater communication overhead overall. Whereas increasing the number of threads from 2 to 3 pays off in any case, which indicates that the loop-level parallelization scales well, the last measurement with 4 threads for each of the 6 MPI processes does not perform well, if DTB is turned on. This is most likely due to the fact that the system is slightly overbooked by 6 times 4 threads plus additional system processes running at the time of this test run on 24 physical CPUs.

Table 2. FLOWer on one Sun Fire 6800, runtime with and without DTB in seconds

#pro-cessors	#MPI processes	#threads	without DTB	with DTB
12	6	2	499.8	350.6
14	7	2	606.9	450.7
16	8	2	511.0	317.0
18	6	3	408.8	286.3
21	7	3	474.3	347.1
24	6	4	363.2	314.0

However, more importantly, the use of the DTB library resulted in substantial improvements in any case. Given that FLOWer is a code used in production in many projects, these results show the benefit of hybrid parallel programming and the performance potential that can be gained from dynamic thread balancing schemes on large SMP nodes.

On the Sun Fire 15K we investigated the behaviour of the ccNUMA architecture. The Solaris 9 operating environment uses the first touch policy by default, if the Memory Placement Optimization (MPO) feature is enabled. The DTB mechanism suffers from the fact, that, if the number of threads is increased for a busy process, an additional thread might be started on a different processor board then slowing down the whole

process, because this thread has to access all its data remotely. As FLOWer uses a static mesh of grid points, our strategy was to turn off the first touch mechanism and use random placement instead for the first phase of the computation. Once the distribution of threads stabilized, we cause all data to move to the very board where it is accessed the next time. Unfortunately this mechanism is by no means standardized. We had to use the MPO APIs provided by Solaris 9.

Figure 1 shows the distribution of threads to the 23 MPI processes over the iteration numbers. Each of these processes starts off with 2 threads, thus occupying 46 processors of the Sun Fire 15K in total. During the first 10 iterations the warm-up phase can clearly be identified. In the following 15 iterations the threads are shifted such that the MPI processes 20, 21, 22, and 23 on the upper part of the graph get 5 threads, whereas the MPI processes depicted on the lower part run with only one thread at the end. At about iteration 30 the DTB library detects that the thread distribution remains stable, triggers the "next touch" mechanism and turns off the steering.

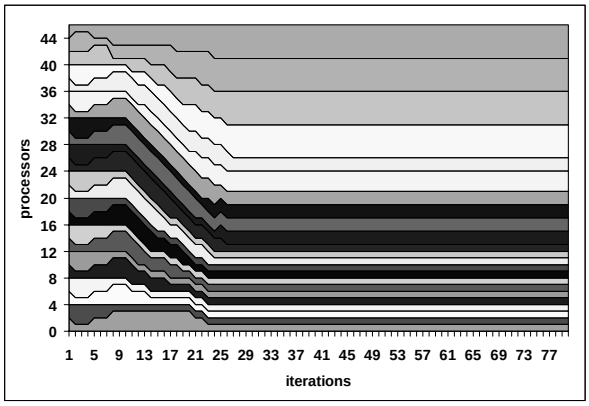


Fig. 1. FLOWer on one Sun Fire 15K: thread distribution over the iteration number

Figure 2 nicely demonstrates the success of this strategy. The number of L2 cache misses satisfied by local and remote memory accesses has been measured with hardware performance counters and is plotted over the runtime. During the first 700 seconds there are by far more remote misses than local misses. Then the "next touch" mechanism is turned on and the DTB steering mechanism is turned off to save overhead and suddenly the situation completely changes. Now the number of local misses is much higher than the number of remote misses. The figure also contains a line indicating the speed of the calculation measured in total GFlop/s. Once the memory is mainly accessed locally the speed increases by about 20 percent.

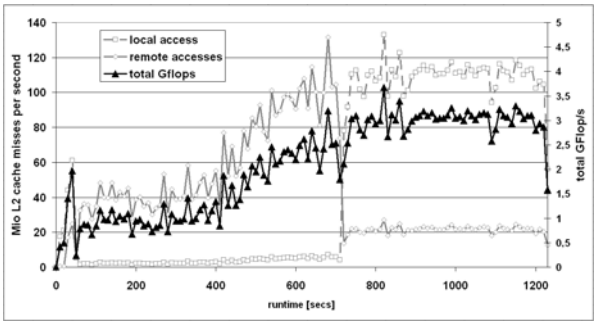


Fig. 2. FLOWer on one Sun Fire 15K, random memory placement at the beginning and then explicit migration after about 700 seconds: local and remote cache misses and GFlop/s

4. Conclusions

By automatically varying the number of threads per MPI process based on timing results of the running program we offer an easy-to-use opportunity to improve the load balance of hybrid applications on clusters of shared-memory machines with fat nodes. The PMPI profiling interface of MPI allows us to easily capture relevant timing information, and as a result, a potential user only has to add a subroutine call in a few places of the code to trigger the dynamic thread balancing (DTB) mechanism.

Essential for the profitability of this technique was the control over the behaviour of the surplus threads such that they no longer block CPUs by spin-waiting. This has been achieved by a Solaris specific environment variable. Here a standardized way of controlling the behaviour of unused threads - for example as an additional parameter of the **`omp_set_num_threads`** runtime function - would be adequate.

Experimental results with the FLOWer CFD solver show that the use of the DTB library results in substantial improvements. The programming effort required to use the DTB library is minimal and it can be employed in a very flexible fashion as the number of additional threads is concerned.

First experiments taking the ccNUMA architecture of the Sun Fire 15K architecture into account revealed very positive results. Means to explicitly move data to where they are used are unfortunately not standardized (for example by an **`omp nexttouch(variable list)`** directive would be helpful), so that we had to employ a low level interface of Solaris 9.

In future work, we will investigate to what extent we can improve the robustness of the DTB mechanism during the warm-up phase and on loaded systems. We also plan to use the library in simulations with adaptive grid refinement.

5. References

1. Jost, G., Jin, H., an Mey, D. and Hattay, F.: Comparing the OpenMP, MPI and Hybrid Programming Paradigms on an SMP Cluster, NAS Technical Report NAS-03-019, NASA Ames Research Center, Moffet Field, CA, Novermber 2003. http://www.nas.nasa.gov/Research/Reports/Techreports/2003/PDF/nas-03-019.pdf

2. Bova, S.W., Breshears, C.P., Gabb, H., Kuhn, B., Magro, B., Eigenmann, R., Gaertner, G., Salvini, S. and Scott, H.: Parallel Programming with Message Passing and Directives. In: Computing in Science and Engineering, September 2001, pp. 22-37.

3. Rabenseifner, R.: Hybrid Parallel Programming on Parallel Platforms. EWOMP'03 - Fifth European Workshop on OpenMP, http://www.rz.rwth-aachen.de/ewomp03/../omptalks/Tuesday/Session7/T01p.pdf.

4. Hesse, M.K., Reinartz, B. and Ballmann, J.: Inviscid Flow Computation for the Shuttle-Like Configuration PHOENIX. Notes on Numerical Fluid Mechanics, Vol. 87, Eds. Chr. Breitsamter, B. Laschka, H.-J. Heinemann, R. Hilbig, Springer 2003, pp. 172-179.

5. Hesse, M.K., Reinartz, B. and Ballmann, J.: Numerical Investigation of the Shuttle-Like Configuration PHOENIX. High Performance Computing in Science and Engineering 2002, Ed. E. Krause, W. JÄager, Springer Verlag, ISBN 3-540-43860-2, pp. 379-390.

6. Hesse, M.K., Reinartz, B. and Ballmann, J.: Numerical Investigation of a Reusable Space Transportation System. In: Proceedings of the 3rd International Symposium on Atmospheric Reentry Vehicles and Systems, Arcachon/France, 24-27 March 2003.

7. Spiegel, A., an Mey, D. and Bischof, C.: Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing, PARA'04 - Workshop on the State-of-the-Art in Scientific Computing, June 2004. Proceedings to be published in the Springer series Lecture Notes in Computer Science

8. Solaris Memory Placement Optimization and Sun Fire Servers, Technical White Paper, March 2003. http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf

OpenMPI — OpenMP like tool for easy programming in MPI

Taisuke Boku[1], Mitsuhisa Sato[1], Masazumi Matsubara[2], Daisuke Takahashi[1]
[1] Graduate School of Systems and Information Engineering, University of Tsukuba
[2] Fujitsu Laboratories
{taisuke,msato,daisuke}@cs.tsukuba.ac.jp, matz@jp.fujitsu.com

1 Introduction

MPI has been still used for programming on wide area of applications and various sizes of problems for parallel programming based on distributed memory architecture though it requires hard labor for application programmers. Since it requires SPMD model programming, it is hard for a user to modify his original sequential program incrementally to MPI version. Of course, it also allows a user to freely describe his program in any style with communication spaghetti for very complicated interaction among parallel processes. However, a large class of scientific applications requires not so complicated communication on data and control on processes, especially for simple data parallel programming.

One of the big reasons of the success in OpenMP is that it allows incremental parallelization for the original sequential code adding appropriate directives. It helps application programmers who are not familiar to parallel processing as well as they can concentrate to the parallelization and optimization on the most time-consuming part in the target codes.

In this research, we propose a programming tool named OpenMPI for OpenMP-like incremental parallelization based on MPI scheme. A user can add directives to his original code to control parallel process invocation and communication among them automatically. The data distribution and collection among parallel processes are semi-automatically performed with invocation of parallel processes and necessary communication functions. It also supports work sharing on parallel loop like OpenMP with auxiliary functions for local index conversion and drawing out information on parallel processes. Moreover, we involve extended functions to control the parallelism dynamically as well as automatic load balancing, introducing a special user-level library named ALB (Automatic Load Balance).

Although OpenMPI limits the flexibility on MPI programming, a wide range of scientific applications may be covered and a restriction with simple notation will greatly help application programmers who want to simply parallelize their target codes. We are currently designing the basic feature of OpenMPI, and will describe several design issues and preliminary specification of it.

2 Motivation

MPI is the most common API for parallel programming on machines with the distributed memory architecture. It supports very flexible parallelization paradigms based on message passing and there is a large set of standard functions included in version 2.0. Although a user can describe his program freely in much optimized style, most of functions are not familiar for typical scientific programmers. In NAS Parallel Benchmark 2.0, for instance, whole set of benchmarks are described only with 26 of MPI version 1.0 functions.

Domain decomposition is the most common scheme for large scale parallel scientific programming on machines with distributed memory architecture. A large set of programs can be described as very simple MPI programs which feature array data distribution, parameter broadcasting, scalar or vector reduction, etc. In such a simple program, however, the user has to describe bothering notation with long arguments to call MPI functions as well as array index conversion which often causes trivial human error.

To help poor scientific programmers who require simple and routine parallelization, we propose a new programming tool named OpenMPI based on the concept of OpenMP, that is, incremental parallelization with simple directives inserted to the original codes. Although the freedom of description is limited by such a style, we believe its feature is enough to describe a large set of programs.

For programming on distributed memory systems, various concepts and systems have been proposed and implemented. HPF[1] is an aggressive and challengeable system for automatic parallelization and data distribution for flexible programming. However, its *template*-based data distribution is too complicated for novice parallel programmers. Its automatic parallelization feature based on owner computation rule is also difficult to exploit the potential parallelism

and performance of the system.

UPC[2] is another challenge for efficient parallel processing on distributed shared memory architecture to provide straight-forward programming for traditional C language. It supports explicit classification of memory attribute of *private* and *shared* to exploit high performance based on process-data annotation and access locality.

There are a number of researches on generic software DSM (distributed shared memory) systems such as TreadMarks[3] or SCASH/SCore[4]. Most of them are based on page-base state management and invalidation-based protocol. Since they require the help of operating system, its performance strongly depends on the application and the system overhead is not negligible on some class of applications.

Compared with these systems, we simply aim to provide simple programming tool for MPI, that is, the user does not expect fully automatic parallelization or the correctness of execution guaranteed by the system. It relieves the painful and bothering programming on MPI as like as OpenMP relieves the programmers from bothering and easy programming error on thread programming.

3 Basic requirements for parallel programming based on domain decomposition

We consider the domain decomposition as the basic parallelization concept in our system. It means that the target problem space is represented as multiple dimensions of data arrays and they are decomposed into multiple chunks to be mapped on to parallel processes. The most time consuming part of the program is the computation on these distributed array data which can be completely parallelized among processes. At certain timing, local or global (collective) communication is required to maintain the consistency and system-wide information sharing.

The basic features required for this purpose are summarized as follows:

array distribution: The arrays to represent the problem space have to be automatically divided into chunks according to the number of parallel processes. To simplify the workload distribution, the target array have to be sliced orthogonally to the selected dimension(s). A decomposed array chunk is allocated to a process and it will not be changed.

index conversion: The global array indexing to the original (sequential) code has to be converted into local array indexing according to the array distribution. By simplified array slicing described above, it can be performed automatically for linear indexing.

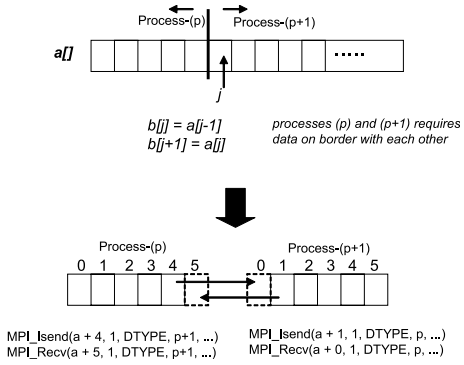


Figure 1. Cross data reference on border elements

data consistency: To support cross-indexing over neighboring elements in an array at the border of distributed array (see Figure 1), automatic data exchanging to maintain data consistency among processes is necessary. It appears as "nearest neighboring communication" in typical MPI programs. To avoid the overhead for dynamic check on data access, the user has to be responsible to the region and timing to keep data consistency.

data reduction: To share global information among processes, collective communications for data reduction are required at certain timing. The target variable and the timing must be explicitly specified by the user.

data synchronization: Since MPI program is described in SPMD manner, even a scalar variable has to be duplicated among parallel processes. A "sequential" computation on such a variable is performed simultaneously in all processes. If the computation depends on some local information kept in a certain process, the computation result must be shared by all processes after the computation. This feature is named "synchronization", and it must be explicitly specified by the user.

auxiliary functions: In OpenMPI, we allow the user to describe explicit MPI function calls as well as the automatically generated MPI code by OpenMPI compiler. However, the user has to understand the data distribution, the size of local array and index range to be assigned to each process. To assist such an explicit data access, miscellaneous functions to provide auxiliary information are required. For instance, index conversion functions to convert global and local array indices help to access an element of local array from a global index written in the original sequential code.

4 OpenMPI directives

The programming with OpenMPI is in similar manner with OpenMP programming. The user inserts appropriate directives to create parallelized version of his program. Here, he must understand that he is describing MPI program with very simple notation, that is, the compiler and run-time system do not provide fully automatic parallelization nor complete shared address space among parallel processes. The user must understand the effect of these directives and be responsible to the result of parallel execution under the rule of OpenMPI directives.

The followings are the typical directives defined in current designing. All directives follow to the notation "**#pragma ompi**". Terms in bold font are reserved words in OpenMPI and terms in italic font are user defined parameters.

- **distvar** (**dim**=*dimension*, **sleeve**=*sleeve_size*)
  The array declared in the next line is distributed on parallel processes. The option **dim** represents the sliced dimension when the array has more than one dimension[1]. The option **sleeve** represents the sleeve size of border elements which may be exchanged automatically (see Figure 1). If the **sleeve** option is omitted, there is no automatic element exchanging on border for that array.

- **global**
  The variable declared in the next line is assumed as global one, that is, the content of the variable will be maintained to the same value among all processes. Since the target architecture is the distributed memory system, it is impossible to make the variable physically coherent. Another directive **sync_var** represents the point to copy the data.

- **for** (**reduction**(*operator* : *variable*))
  The for loop at the next line is parallelized and all distributed array data indexed by the control function of **for** statement and its linear expression are referred only within the assigned region to the process according to slicing feature of **distvar** directive. The option **reduction** causes the collective communication on *variable* with *operator* after the end of the loop.

- **sync_sleeve** (**var**=*variable list*)
  At this point, the sleeve data of specified distributed array are exchanged to maintain the correctness of neighboring data reference. If **var** option to specify the name of array is omitted, all distributed array data are processed.

- **sync_var** (**var**=*variable list*,**master**=*node_id*)
  The contents of the specified global variable (scalar or array) among all processes are synchronized by copying a master data to others. Process-0 is assumed as the default master if no **master** option is specified. Otherwise, the specified node is the master.

- **single** (**master**=*node_id*)
  The next block is executed by only one process as the representative of all processes. The process to execute the following block is specified by **master** option. Otherwise, process-0 is assumed as the master.

[1]Currently, only one dimension can be sliced, and there is no feature of multiple-dimension slicing.

5 Sample code

Figures 2 and 3 show a sample code of OpenMPI program in C. The target problem is the explicit solution of 2-dimensional Laplace equation with convergence check. 2-D arrays `u` and `nu` are sliced at the second dimension (X-dimension, *dim=1*) as shown in Figure 4.

```
#include <openmpi.h>
#include <stdio.h>

#define XSIZE 128
#define YSIZE 128
#define EPS        1.0e-2

#pragma ompi distvar (dim=1,sleeve=1)
double u[YSIZE + 2][XSIZE + 2];
#pragma ompi distvar (dim=1)
double nu[YSIZE + 2][XSIZE + 2];

main()
{
  int i, j, k;
  double res1, res2, tmp;
  int itr;

  /* initialization */
  for(i = 1; i <= YSIZE; i++)
#pragma ompi for
    for(j = 1; j <= XSIZE; j++)
      u[i][j] = 1.0;
#pragma ompi for
  for(j = 1; j <= XSIZE; j++){
    u[0][j] = 10.0;
    u[YSIZE+1][j] = 10.0;
  }
  for(i = 1; i <= YSIZE; i++){
    u[i][0] = 10.0;
    u[i][XSIZE+1] = 10.0;
  }
  /* iteration */
  itr = 1;
  while(1){
#pragma ompi sync_sleeve
    for(i = 1; i <= YSIZE; i++)
#pragma ompi for
    for(j = 1; j <= XSIZE; j++)
      nu[i][j] = (u[i-1][j] + u[i+1][j]
                + u[i][j-1] + u[i][j+1]) / 4.0;
    /* convergence check */
    res1 = 0.0;
```

Figure 2. Example code (Laplace Eq.) (1/2)

```
    for(i = 1; i <= YSIZE; i++){
      res2 = 0.0;
#pragma ompi for reduction(+:res2)
      for(j = 1; j <= XSIZE; j++){
        tmp = (nu[i][j] - u[i][j]) / u[i][j];
        res2 += tmp * tmp;
      }
      res1 += res2;
    }
#pragma ompi single
    fprintf(stderr, "itr=%d res=%g\n", itr, res1);
    /* update */
    for(i = 1; i <= YSIZE; i++)
#pragma ompi for
      for(j = 1; j <= XSIZE; j++)
        u[i][j] = nu[i][j];
    if(res1 < EPS)
      break;
    itr++;
  }

  /* output */
  for(i = 1; i <= YSIZE; i++)
#pragma ompi for
    for(j = 1; j <= XSIZE; j++)
      printf("%d %d %f\n", i, j, u[i][j]);
}
```

Figure 3. Example code (Laplace Eq.) (2/2)

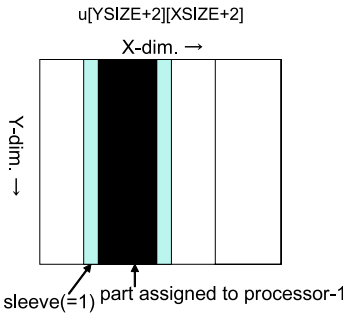


Figure 4. 2-D array distribution of *u*

A subpart of array `u` assigned to each process has two sleeves on left and right edges with size=1. It means that these edges at the border of neighbors are overlapped with neighbors' data on corresponding edges with the width of 1. These overlapped data are automatically exchanged at the point of "**#pragma ompi sync_sleeve**" in every iteration. Local array of `u` is declared including these two additional elements to accept data sent from left and right neighbors, then they can be referred in the following second loop to calculate new value of `u` saved into `nu`. On the other hand, `nu` has no sleeve because subparts of `nu` are completely treated as local variable without any interaction with neighbors.

In the latter half of the big loop (while statement), a scalar variable `res2` is treated as reduction variable on for loop to check the convergence of the result.

6 Compiler

The compiler of the prototype implementation of Open-MPI is now under construction. It is implemented as a source-to-source converter to generate a complete MPI code from the target code. Figures 5 and 6 show the image of converted code of the program shown in Figures 2 and 3.

```
#include "openmpi.h"
#include <mpi.h>
#include <stdio.h>

#define XSIZE 128
#define YSIZE 128
#define EPS        1.0e-3

double u[YSIZE + 2][XSIZE + 2];
double nu[YSIZE + 2][XSIZE + 2];

/* variables inserted by compiler */
int _nproc, _myid;
int _u_idx_lower, _u_idx_upper, _u_idx;
int _nu_idx_lower, _nu_idx_upper, _nu_idx;
int _u_sleeve = 1;
int _u_dim = 1;
int _nu_sleeve = 0;
int _nu_dim = 1;

main(int argc, char **argv)
{
  int i, j, k;
  double res1, res2, tmp;
  int itr;
  int _l_j, _l_j_llimit, _l_j_ulimit;
                              /* local index var. */
  double _tmp_res2;           /* for reduction */

  ompi_init(&argc, &argv);
  ompi_record_var(u, 2, 1, 1, sizeof(double),
       &_u_idx_lower, &_u_idx_upper, YSIZE+2, XSIZE+2);
  ompi_record_var(nu, 2, 1, 0, sizeof(double),
       &_nu_idx_lower, &_nu_idx_upper, YSIZE+2, XSIZE+2);
  /* initialization */
  for(i = 1; i <= YSIZE; i++){
```

Figure 5. Converted code of Laplace Eq. (1/2)

```
    for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
      u[i][_l_j] = 1.0;
  }
  _l_j_llimit = ompi_get_llimit(1, _u_idx_lower);
  _l_j_ulimit = ompi_get_ulimit(XSIZE, _u_idx_upper);
  for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++){
    u[0][_l_j] = 10.0;
    u[YSIZE+1][_l_j] = 10.0;
  }
  for(i = 1; i <= YSIZE; i++){
    _u_idx = ompi_convert_local(0, _u_idx_lower,
                                _u_idx_upper);
    if(_u_idx >= 0){
      u[i][_u_idx] = 10.0;
    }
    _u_idx = ompi_convert_local(XSIZE+1, _u_idx_lower,
                                _u_idx_upper);
    if(_u_idx >= 0){
      u[i][_u_idx] = 10.0;
    }
  }
  /* iteration */
  while(1){
    ompi_sync_sleeve();
    for(i = 1; i <= YSIZE; i++){
      _l_j_llimit = ompi_get_llimit(1, _u_idx_lower);
      _l_j_ulimit = ompi_get_ulimit(XSIZE,
                                    _u_idx_upper);
      for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit;
          _l_j++)
        nu[i][_l_j] = (u[i-1][_l_j] + u[i+1][_l_j]
              + u[i][_l_j-1] + u[i][_l_j+1]) / 4.0;
    }
    /* convergence check */
    res1 = 0.0;
    for(i = 1; i <= YSIZE; i++){
      res2 = 0.0;
      _tmp_res2 = 0.0;
      _l_j_llimit = ompi_get_llimit(1, _u_idx_lower);
      _l_j_ulimit = ompi_get_ulimit(XSIZE, _u_idx_upper);
      for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit;
          _l_j++){
        tmp = (nu[i][_l_j] - u[i][_l_j]) / u[i][_l_j];
        _tmp_res2 += tmp * tmp;
      }
      ompi_reduce(OMPI_REDUCE_OP_SUM, OMPI_DATA_DOUBLE,
                  1, &res2, &_tmp_res2);
      res1 += res2;
    }

    if(ompi_master()){
      fprintf(stderr, "itr=%d  res=%g\n", itr, res1);
    }
    /* update */
    for(i = 1; i <= YSIZE; i++){
      _l_j_llimit = ompi_get_llimit(1, _u_idx_lower);
      _l_j_ulimit = ompi_get_ulimit(XSIZE, _u_idx_upper);
      for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
        u[i][_l_j] = nu[i][_l_j];
                      /* possible, not aligned ... */
    }
    if(res1 < EPS)
      break;
    itr++;
  }

  /* output */
  for(i = 1; i <= YSIZE; i++){
    _l_j_llimit = ompi_get_llimit(1, _u_idx_lower);
    _l_j_ulimit = ompi_get_ulimit(XSIZE, _u_idx_upper);
    for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
      printf("%d %d %f\n", i,
         ompi_convert_global(_l_j, _myid), u[i][_l_j]);
  }
}
```

Figure 6. Converted code of Laplace Eq. (2/2)

Array variables to be distributed to all processes are declared in the same manner as the original code. Currently, only global and static variables are allowed as **distvar** objects, and the array size of converted version is as same as the original. It is because the optimal size of local array, which could be smaller than the original, depends on the number of processes to share the array, and we consider that the number of processors to be used is fixed at run time. One of the solutions is to assume the minimum number of processes at compile time and reduce the size of array.

According to the index range assigned to the process, the reference to arrays is limited between the lower and upper limits precalculated by internal functions `ompi_get_llimit()` and `ompi_get_ulimit()`. As shown in this code, only a linear expression of control variable (`j`, here) is allowed for OpenMPI *for* statement.

At the corresponding point of **sync_sleeve** directive, an international function `ompi_sync_sleeve()` is called to exchange sleeve data between neighboring processes. Within the function, the array information (starting address, dimension, size of each dimension and sleeve size) is referred to determine the data area to be sent and received. For this purpose, all distributed array data are registered by an internal function `ompi_record_var()` at the beginning of main program. The contents of these sleeves are packed into the send buffers, then they are transferred asynchronously. After receiving, all received data are unpacked into the corresponding sleeve area in the target arrays. Since we allow any dimension of the distributed array to be sliced, these exchanged area generally forms a block-stride pattern. Therefore, we need a constant size of pack/unpack buffer for each sleeve area.

7 Automatic Load Balancing

Since OpenMPI introduces the abstraction of parallelization instead of explicit MPI parallel programming, there is a possibility to control the parallelism not only at compile time but also at run time. Ordinary MPI code assumes that the number of parallel processes is fixed at the beginning of run time, but MPI 2.0 supports the dynamic increase/decrease of processes while executing the code. Applying this feature to our system, we can dynamically control the parallelism while iterating the loop. For instance, if we operate a big PC cluster for multiple jobs, this dynamic feature greatly enhances the total system efficiency according to the size and number of jobs to be executed. With explicit MPI programming, it is very hard to describe such a dynamic code even with MPI 2.0 feature.

OpenMPI compiler generates a pure MPI code which can be executed on generally used MPI libraries such as LAM[5] or MPICH[6]. We are also designing an optional feature to support dynamic parallelism based on LAM MPI

7.0 or higher. On LAM MPI, there is a set of low level commands to control the number of running parallel processes on user level. For easier implementation and flexibility on user interface, we apply ALB (Automatic Load Balance) tool, which is developed by Fujitsu Laboratories, to OpenMPI compiler.

ALB is developed based on dynamic parallelization control feature on LAM MPI. The tool consists of two parts: API library for extended MPI programming and user level commands to control the number of processors involved in parallel execution. Current design of OpenMPI compiler is strongly considered about the suitability of the converted code with ALB library. For example, array distribution with one dimensional slicing with sleeve data and the synchronization of global variable are directly supported by ALB. Moreover, ALB checks the increase/decrease of processes at the timing of synchronization among variables, and automatically redistributes registered arrays according to the information registered at the beginning of computation.

In this extended abstract, we have no space to describe the detail of ALB, however the combination of ALB-based OpenMPI and run-time monitoring system to check the load of the cluster provides a flexible and efficient system management method for large scale cluster.

8 Conclusions

OpenMPI is a new scheme to provide very easy programming method of MPI code for scientific application users. Although it limits the flexibility and freedom of programming, the concept of incremental parallelization of OpenMP is partially applied to help programmers. ALB also supports the dynamic parallelism control on OpenMPI code for efficient usage of CPU resources in a cluster system shared by multiple users and tasks. The easiness of ALB usage is greatly enhanced by introducing OpenMPI.

The system is still under construction. After the completion, we will describe various sample codes including NAS PB to demonstrate the capability of code description and parallelization of OpenMPI.

References

[1] http://dacnet.rice.edu/Depts/CRPC/HPFF/

[2] http://upc.gwu.edu/

[3] C. Amza, et. al., “TreadMarks: Shared Memory Computing on Networks of Workstations”, IEEE Computer, Vol. 29, No. 2, pp. 18-28, February 1996.

[4] http://www.pccluster.org/score/dist/score/html/en/reference/scash/index.html

[5] http://www.lam-mpi.org/

[6] http://www-unix.mcs.anl.gov/mpi/mpich/

Session 6: OpenMP tools 2

Chair: Federico Massaioli, CASPUR, Italy

- M. Müller, C. Niethammer, B. Chapman, Y. Wen, and Z. Liu, Validating OpenMP 2.5 for Fortran and C/C++
- G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schultz, Checkpointing Shared Memory Programs at the Application-level
- J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguadé, and J. Labarta, Nanos Mercurium: a Research Compiler for OpenMP

Validating OpenMP 2.5 for Fortran and C/C++

Matthias Müller, Christoph Niethammer
HLRS, University of Stuttgart, Allmandring 30, D-70550 Stuttgart, Germany,
mueller@hlrs.de

Barbara Chapman, Yi Wen, Zhenying Liu
University of Houston

Abstract

We present a collection of C/C++ and Fortran programs with OpenMP directives that were designed to validate the correctness of an OpenMP implementation. The validation methodology and implemented tests are demonstrated. We also discuss the differences between the Fortran and C validation suite and extensions made possible by the clarifications introduced by OpenMP 2.5.

1. Introduction

In the last years OpenMP [3] has found wide spread acceptance as a portable programming model. This is not only reflected by the use of OpenMP in many applications but also by the constant development of the standard. Since its first release in 1997, the OpenMP Architecture Review Board has released several versions of the standard. The OpenMP 2.5 standard is currently being finalized. It will not only merge the Fortran and C/C++ language binding, but it also contains a number of clarifications or more detailed specifications. In [2] a first proposal for an open and portable validation suite for the C language binding was presented. In this paper we discuss its extension to Fortran, improvements made during the last year and new tests that were made possible due to the clarifications introduced with the upcoming OpenMP 2.5.

This paper is organized as follows. In the next section we describe the basic design of the validation suite. In Section 3 we describe the improvements of the suite since last year and explain how it has been extended. Section 4 shows some results with current compilers. We draw a conclusion in Section 5.

2. Design

The idea of the suite is to have a subroutine for each OpenMP construct that returns the value *true* if the construct works as expected and *false* otherwise. The target is to completely cover all constructs and clauses of OpenMP (see Fig. 1).

Each subroutine will perform a calculation for which the correct result should depend on the correct functionality of the used OpenMP construct. In order to check the dependency of the result from the correct implementation we evaluate the result when the construct is missing. To increase the likelihood of catching a race condition we perform N repetitions of each test, currently N is fixed and set to 20. Clearly, if a test fails once the construct is not working correctly. To estimate the likelihood that the test is passed accidentally we take the following approach: If n_f is the number of failed cross checks and M the total number of iterations, we estimate the probability of the test to fail with $p = \frac{n_f}{M}$. The probability that an incorrect implementation passes the test is $p_a = (1-p)^M$ and the certainty of the test $p_c = 1 - p_a$.

It is clear that a missing OpenMP directive will lead to different race conditions than a broken implementation, but the approach described above gives an estimate of the reliability of the test. The existence of a race condition will also depend on the optimization level, e.g. if a variable is stored in a register, a missing `private` clause may not lead to wrong results.

Of course a directive is not simply removed for the crosscheck, but replaced by a directive that does not contain the functionality to be tested. For example, when a `firstprivate` is removed for the crosschecks, it is not merely removed, but replaced by a `private` clause which will change the initial values of the variables to be undefined. Tab. 1 contains a list of directives and their replacements.

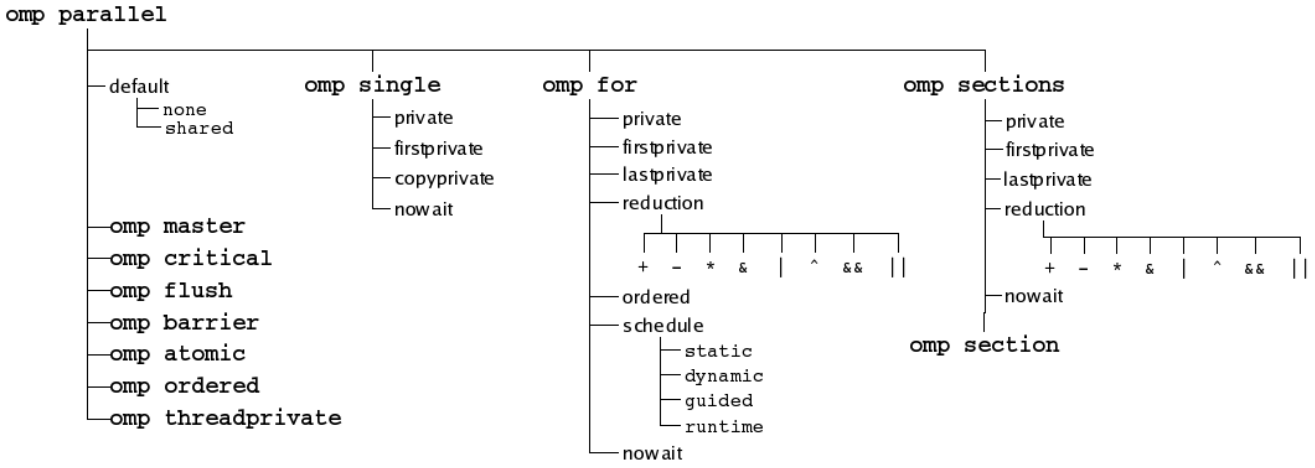


Figure 1. An overview of OpenMP directives and clauses.

Clause	Substitution
if	
private	shared
firstprivate	private
lastprivate	private
ordered	
single	
copyprivate	private
copyin	
reduction	
num_threads	

Table 1. OpenMP directives and clauses and their substitutions in the cross checks.

3. Extensions of the Validation Suite

An early C version of the validation suite was introduced in [2], and we extend it in this section.

3.1. Extension to Fortran

We realized the Fortran OpenMP validation suite by converting the original C version [2] to Fortran. In almost all cases, this translation was straightforward. However, we did have to modify several variable names that were not accepted by some OpenMP compilers. Fortran 90 was used as it gave us the convenience of long procedure names and enabled us to give these names that are similar to their C counterparts [2]. Furthermore, dynamic arrays and the derived type of Fortran 90 are used in accordance with C. We sometimes force the lower bound of an array to be 0 for convenience although by default it is 1 in Fortran. As a result, the suite currently does not work with Fortran 77; however, it is easy to produce equivalent Fortran 77 and this will also be provided. One exception is the strategy used to test OpenMP locks, which employs the Fortran 90 KIND construct. A lock variable with the `KIND` of `OMP_LOCK_KIND` and/or `OMP_NESTED_LOCK_KIND`, declared in *omp_lib.h*, must be specified for the current test.

```
    USE OMP_LIB
    INTEGER (KIND=OMP_LOCK_KIND) :: LCK
    CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK)
    ...
    CALL OMP_SET_LOCK(LCK)
    ...
    CALL OMP_UNSET_LOCK(LCK)
!$OMP END PARALLEL
  ...
    CALL OMP_DESTROY_LOCK(LCK)
```

At this time, our validation suite has been used to test all of the Fortran OpenMP 1.0 features for Sun Forte 7 Fortran 95 compiler, Intel 8.0 compiler and the Open64 compiler [1]. We are working on the complete Fortran OpenMP 2.0 and 2.5 specification, including `WORKSHARE` directive.

3.2. Reductions

In the previous version of our tests, the `reduction` clause was tested only by calculating $S = \sum_{i=1}^{N} i$ and comparing it with the known result $S = N * (N + 1)/2$. Both N and i were integers to avoid any problems with rounding errors. The test was limited to this single calculation; other operations or types of variables were not tested. In the meantime we have implemented tests for all

reduction operations (+,-,*,,&,|,^,&&,|| in C and +,*,-,.AND.,.OR.,.EQV.,or.NEQV. in Fortran) , both for integer as well as floating point variables. The input data sets for the boolean operations are constructed to detect any omissions of a single value. E.g. for the logical or all values of the input array are false with only one value set to true.

For the reduction operation of floating point variables we use the calculation of $E = \sum_{i=1}^{N} \frac{1}{k^i}$ with $k = 3$ and compare it with the analytical result.

3.3. Schedule clauses

One of the areas of clarifications introduced with OpenMP 2.5 are the scheduling clauses. Now, several tests are possible with respect to the schedule clause used in conjunction with a parallel loop. For static as well as dynamic scheduling we check whether the chunks have the requested size, with the legal exception of the last chunk. For the guided clause we verify that the chunk size is decreasing following the algorithm outlined in the OpenMP specification. For schedule(runtime) the tests of the respective clause apply, and the execution environment (see Sec. 3.5) has to set the correct value for OMP_SCHEDULE.

The basic idea for identifying the chunks is simple. We simply perform the loop

```
#pragma omp for schedule(runtime)
    for(i=0;i<MAX_SIZE;++i)
    {
        a[i]=thread_id;
    } /*end omp for*/
```

However, in the dynamic case we must avoid assigning two successive chunks to the same thread, because this will appear like one larger chunk in the later analysis. The execution flow of the parallel loop is under the control of the compiler and runtime environment. Control is handed to the user program only after the chunk is already assigned to a specific thread. During execution only loop iterations are visible to the threads, so there is no way to decide if a single iteration is at the beginning or end of a chunk.

The algorithm we implemented is based on the idea that each thread has to wait until at least one different thread has been assigned the next chunk. The highest iteration count that is reached by the threads is saved in the shared variable *maxiter*. Each thread waits before it continues to iterate until

1. another thread has reached a higher iteration count
2. a timeout occurs
3. the first thread has left the parallel loop

The first condition makes sure that two successive chunks cannot be assigned to the same thread. The second guarantees progress, e.g. when the last chunk is assigned to the threads. The last condition combined with a nowait clause improves the execution time, because no timeout has to be used at the end of the loop.

With this test we can now not only test the standard conformance of the compilers, but also what decision the implementors have made regarding implementation defined behavior. Unfortunately the tests only worked with seven different compilers. One did reject the source code, and with two it did not work correctly. This failure was expected in once case, because the flush test also did not work correctly.

The first implementation defined behavior is the default schedule. All compiler used static as their default schedule. Another option is the distribution of loop iterations if the total iteration count is not a multiple of the chunksize. One option is to reduce the last chunk, another option is to reduce the last chunk assigned to each thread. Five compiler take the first option and two compilers the last. For the guided scheduling the OpenMP 2.5 standard requires that the chunksize is proportional to the number of remaining items divided by the number of threads. The linear factor is undefined. We found that the compilers either choose 1.0 or 0.5 for the factor. One compiler did not implement guided correctly, all chunks had the same size.

3.4. Test Output

As the version before the new validation suite shows a brief overview of the test results during execution. New to this version is the creation of a logfile, which contains detailled information on the tests. This logfile contains the result of all tests that helps to find out much better what has been going wrong. For example the test for the reduction clause contains several smaller tests of its options. So if only one test within the reductiontest failed you see in the overview that the whole test failed but in the logfile you can see exactly which option didn't pass it's test and what was the reason for this.

3.5. Execution Environment

We have begun to create a flexible environment for building and executing the tests in the validation suite with the goal of ensuring that neither compilation nor execution of the validation suite will be terminated by the occurrence of an error; an additional goal is to permit users to add or remove individual tests, and set compiler options, as desired. Furthermore, we intend to provide the test results in a user friendly way. The environment needs components to control both the compilation and execution of the

OpenMP compiler tests. In the current version of the validation suite, if a compiler error is encountered in the translation of an OpenMP construct, directive or library function, then the compilation process will stop and the executables for the validation suite will not be created. In order to solve this problem, we intend to use a script that will enable the compiler to continue to translate tests for the remaining OpenMP language features and library routines in the presence of such errors. We use a Perl script to accomplish this functionality as Perl is an interpreted language that is convenient for reading, extracting and writing text files. Perl is also suitable for executing system commands, such as invoking compilation, and handling the return information from them. Hence Perl will enable us to recover from compiler errors and generate a report in this situation. The Perl script acts as a driver for the OpenMP compiler being tested, invoking it to compile the individual tests that are part of the suite; while doing so, it keeps a record of any compiler errors that are encountered. The generated code will be executed under the control of a Perl script at runtime in order to complete the execution of all the desired tests. Thus deadlocks may be detected and reported on if a particular test lasts very much longer than expected. The portability of this environment will be a strong consideration in our implementation of the Perl scripts.

The environment will enable the validation suite to be used to test a particular language feature of OpenMP if desired. Users will be permitted to input the name of a certain directive which is to be tested individually. In particular, after users have obtained a complete report of their OpenMP compiler, they may want to retest a specific language feature that was not passed by the validation suite. On the other hand, we intend to group some similar tests together and give them a name so that users may specify, for example, *worksharing* as a means to request the environment to build and run precisely those tests that evaluate the implementation of worksharing constructs in OpenMP, including DO, SECTIONS, SINGLE and WORKSHARE. We also plan to allow users to select OpenMP 1.0 or OpenMP 2.0 so all the features for the given version of the API can be tested accordingly.

We use different files to store information internally and output results in a user friendly format. The main purpose is naturally to report which directives pass or fail the test. We need a configuration file to store the executable path for the compiler and the compiler flags specified by the user. Note that different compiler flags may result in different outputs, and it might be interesting or necessary to evaluate an OpenMP compiler using different sets of flags. A log file is necessary to save information when a compiler error occurs. Once a particular directive or set of directives is selected for testing, we will look up an *information file* that contains details of the separate tests and test sequences. This file includes a table with directive names, function names and file names. *Function names* refer to the functions that are implemented to test the directives, and *file name* refers to the files that include these functions. This information file may also contain addition information, especially for human understanding. After compilation, each test is executed independently and the outputs are written to a file. We will include a discussion of this environment in our presentation at EWOMP 2004.

4. Results

Table 2 shows a summary of the results with ten different compilers. A -1 indicates that the compiler failed the test. A positive number indicates the effectiveness of the cross checks. If the result is 40 it means that all test passed and 40 % of the cross checks failed. For this test we have chosen 20 iteration for the tests and cross checks.

The results show that the validation suite is very portable. The only exception is one compiler which refuses parts of the code. The compiler claims that the ‘Loop is not countable’, due to the constructs used to control the execution, like described in Sec. 3.3. In Table 2 the affected tests are indicated as failed.

5. Conclusion

In this paper we presented an OpenMP validation suite suitable to test an OpenMP implementation for the C and Fortran language binding. Compared to the previous version the suite was not only extended to cover Fortran, but a large number of tests have been added to cover the complete specification of OpenMP. Some of the added tests have only been possible with the added clarifications of OpenMP 2.5.

References

[1] Open64 compiler tools. `http://open64.sourceforge.net`.

[2] M. S. Müller and P. Neytchev. An OpenMP validation suite. In *Fifth European Workshop on OpenMP*, Aachen University, Germany, Sept. 2003.

[3] OpenMP Architecture Review Board. *OpenMP Specifications*. `http://www.openmp.org/specs`.

Compiler	1	2	3	4	5	6	7	8	9	10
check_has_openmp	100	100	100	100	100	100	100	100	100	100
check_omp_get_num_threads	100	100	100	100	100	100	100	100	100	100
check_omp_in_parallel	100	100	100	100	100	100	100	100	100	100
for ORDERED	100	100	100	100	100	100	100	100	100	100
for REDUCTION	100	100	100	100	100	100	100	100	100	100
for PRIVATE	-1	100	100	70	100	100	100	100	95	100
for FIRSTPRIVATE	0	100	100	100	100	55	100	100	100	0
for LASTPRIVATE	100	100	100	55	0	5	100	0	70	100
section PRIVATE	0	100	95	0	100	100	100	100	100	95
section FIRSTPRIVATE	100	100	100	100	100	100	100	95	100	100
section LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
SINGLE	100	100	100	100	100	100	100	100	100	100
single PRIVATE	40	0	45	15	10	0	0	0	0	0
SINGLE NOWAIT	100	100	100	100	100	100	100	100	100	100
parallel for ORDERED	100	100	100	100	100	100	100	100	100	100
parallel for REDUCTION	100	100	100	100	100	100	100	100	100	100
parallel for PRIVATE	0	100	100	55	100	100	100	100	100	100
parallel for FIRSTPRIVATE	100	100	100	100	100	100	100	100	100	100
parallel for LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
parallel section PRIVATE	0	100	100	0	100	100	100	100	100	5
parallel section FIRSTPRIVATE	100	100	100	100	100	100	100	95	100	100
parallel section LASTPRIVATE	100	100	100	100	100	100	100	100	100	100
omp MASTER	100	100	100	100	100	100	100	100	100	100
omp CRITICAL	35	100	100	0	15	100	100	100	100	100
omp ATOMIC	60	95	100	0	35	100	100	100	100	5
omp BARRIER	100	100	100	100	100	100	-1	100	100	100
omp FLUSH	0	0	0	0	0	0	-1	0	-1	0
omp THREADPRIVATE	100	100	100	100	-1	100	-1	100	100	100
omp COPYIN	100	100	100	100	-1	95	-1	100	100	100
omp LOCK	-1	100	100	20	100	85	100	100	100	100
omp TESTLOCK	-1	100	100	0	100	90	100	100	100	100
omp NEST_LOCK	-1	100	100	65	100	65	100	100	100	100
omp NEST_TESTLOCK	-1	100	100	20	100	80	100	100	100	100
Summary	fail	pass	pass	pass	fail	pass	fail	pass	fail	pass

Table 2. Results of the cross checks. A -1 indicates the failure of a test.

Checkpointing Shared Memory Programs at the Application-level

Greg Bronevetsky, Daniel Marques, Keshav Pingali*
Department of Computer Science
Cornell University
Ithaca, NY 14853
{bronevet,marques,pingali}@cs.cornell.edu

Peter Szwed
School of Electrical and Computer Engineering
Cornell University
Ithaca, NY 14853
pkszwed@csl.cornell.edu

Martin Schulz†
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
schulzm@llnl.gov

ABSTRACT

Trends in high-performance computing are making it necessary for long-running applications to tolerate hardware faults. The most commonly used approach is checkpoint and restart (CPR) - the state of the computation is saved periodically on disk, and when a failure occurs, the computation is restarted from the last saved state. At present, it is the responsibility of the programmer to instrument applications for CPR.

Our group is investigating the use of compiler technology to instrument codes to make them self-checkpointing and self-restarting, thereby providing an automatic solution to the problem of making long-running scientific applications resilient to hardware faults. Our previous work focused on message-passing programs.

In this paper, we describe such a system for shared-memory programs running on symmetric multiprocessors. This system has two components: (i) a pre-compiler for source-to-source modification of applications, and (ii) a runtime system that implements a protocol for coordinating CPR among the threads of the parallel application. For the sake of concreteness, we focus on a non-trivial subset of OpenMP that includes barriers and locks.

One of the advantages of this approach is that the ability to tolerate faults becomes embedded within the application itself, so applications become self-checkpointing and self-restarting on any platform. We demonstrate this by showing that our transformed benchmarks can checkpoint and restart on three different platforms (Windows/x86, Linux/x86, and Tru64/Alpha). Our experiments show that the overhead introduced by this approach is usually quite small; they also suggest ways in which the current implementation can be tuned to reduced overheads further.

*This research was supported by DARPA Contract NBCH30390004 and NSF Grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

†Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

1. INTRODUCTION

The problem of making long-running computational science programs resilient to hardware faults has become critical. This is because many computational science programs such as protein-folding codes using *ab initio* methods are now designed to run for weeks or months on even the fastest available computers. However, these machines are becoming bigger and more complex, so the mean time between failures (MTBF) of the underlying hardware is becoming less than the running times of many programs. Therefore, unless the programs can tolerate hardware faults, they are unlikely to run to completion.

The most commonly used approach in the high-performance computing arena is checkpoint and restart (CPR). The state of the program is saved periodically during execution on stable storage; when a hardware fault is detected, the computation is shut down and the program is restarted from the last checkpoint. Most existing systems for checkpointing such as Condor [12] take System-Level Checkpoints (SLC), which are essentially core-dump-style snapshots of the computational state of the machine. A disadvantage of SLC is that it is very machine and OS-specific. Furthermore, system-level checkpoints by definition cannot be restarted on a platform different from the one on which they were created.

In most programs however, there are a few key data structures from which the entire computational state can be recovered; for example, in an n-body application, it is sufficient to save the positions and velocities of all the particles at the end of a time step. In Application-Level Checkpointing (ALC), the application program is written so that it saves and restores its own state. This has several advantages. First, applications become self-checkpointing and self-restarting, eliminating the extreme dependence of SLC implementations on particular machines and operating systems. Second, if the checkpoints are created appropriately, they can be restarted on a different platform. Finally, in some applications, the size of the saved state can be reduced dramatically. For example, for protein-folding applications on the IBM Blue Gene machine, an application-level checkpoint is a few megabytes in size whereas a full system-level checkpoint is a few terabytes. For applications on most platforms, such as the IBM Blue Gene and the ASCI machines, hand-implemented ALC is

the default.

In this paper, we describe a semi-automatic system for providing ALC for shared-memory programs, particularly in the context of Symmetric Multi-Processor (SMP) systems. Applications programmers need only instrument a program with calls to a function called `potentialCheckpoint()` at places in the program where it may be desirable to take a checkpoint (for example, because the amount of live state there is small). Our Cornell Checkpointing Compiler (C^3) tool then automatically instruments the code so that it can save and restore its own state. We focus on shared-memory programs written in a subset of OpenMP [22] including parallel regions, locks, and barriers. We have successfully tested our checkpoint/restart mechanism on a variety of OpenMP platforms including Windows/x86 (Intel compiler), Linux/x86 (Intel compiler), and Tru64/Alpha (Compaq/HP compiler).

The system described here builds on our previous work on ALC for message-passing programs [5, 4]. By combining the shared-memory work described here with our previous work on message-passing programs, it is possible obtain fault tolerance for hybrid applications that use both message-passing and shared-memory communication.

The remainder of this paper is structured as follows. In Section 2, we briefly discuss prior work in this area. In Section 3, we introduce our approach and how our tool is used. In Section 4, we present experimental results. Finally, we discuss ongoing work in Section 5.

2. PRIOR WORK

Alvisi et al. [10] is an excellent survey of techniques developed by the distributed systems community for recovering from fail-stop faults.

The bulk of the work on CPR of parallel applications has focused on message-passing programs. Most of this work deals with SLC approaches, such as [25] [6] and thus results in solutions where the message passing library must be modified in order to allow checkpointing to take place. At the application-level, most solutions are hand-coded checkpointing routines run at global barriers. Recently, our research group has pioneered preprocessor-based approaches for implementing ALC (semi-)automatically [5, 4].

Checkpointing for shared memory systems has not been studied as extensively. The main reason for this is that shared memory architectures were traditionally limited in their size and hence fault tolerance was not a major concern. With growing system sizes, the availability of large-scale NUMA systems, and the use of smaller SMP configurations as building blocks for large-scale MPPs, checkpointing for shared memory is growing in importance.

Existing approaches for shared memory have been restricted to SLC and are bound to particular shared memory implementations. Both hardware and software approaches have been proposed. SafetyNet [23] is an example of a hardware implementation. It inserts buffers near processor caches and memories to log changes in local processor memories as well as messages between processors. While very efficient (SafetyNet can take 10K checkpoints per second), SafetyNet requires changes to the system hardware and is therefore not portable. Furthermore, because it keeps its logs inside regular RAM or at best battery-backed RAM rather than some kind of stable storage, SafetyNet is limited in the kinds of failures it is capable of dealing with.

On the software side, Dieter et al. [8] and the *Berkeley Labs Linux Checkpoint/Restart* [9] provide checkpointing for SMP systems. Both approaches modify specific systems and are thus bound to them, rendering these solutions non-portable.

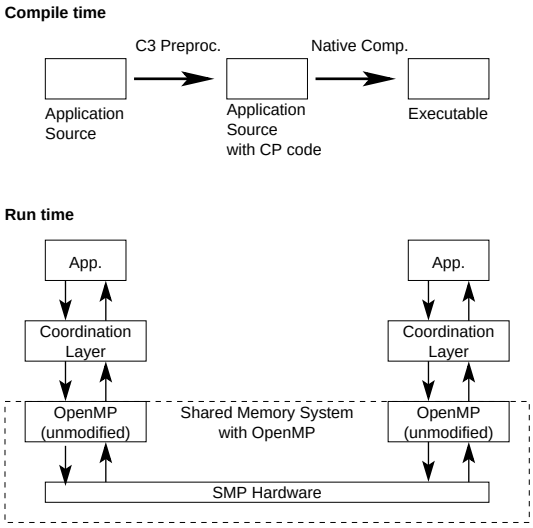


Figure 1: Overview of the C^3 system.

In addition, several projects have explored checkpointing for software distributed shared memory (SW-DSM) [13, 21]. They are all implemented within the SW-DSM system itself and exploit internal information about the state of the shared memory to generate consistent checkpoints. They are therefore also bound to a particular shared memory implementation and do not offer a general and portable solution.

3. OVERVIEW OF APPROACH

Figure 1 describes our approach. The C^3 pre-compiler reads C/OpenMP application source files and instruments them to perform application-level saving of shared and thread-private state. The only modification that programmers must make to source files is to insert calls to a function called `potentialCheckpoint()` at points in the program where a checkpoint may be taken. Ideally, these should be points in the program where the amount of live state is small.

It is important to note that checkpoints do not have to be taken every time a `potentialCheckpoint()` call is reached; instead, a simple rule such as "checkpoint only if a certain quantum of time has elapsed since the last checkpoint" is used to decide whether to take a checkpoint at a given location. Checkpoints taken by individual threads are kept consistent by our coordination protocol.

The output of the pre-compiler is compiled with the native compiler on the hardware platform, and linked with a library that implements a coordination layer for generating consistent snapshots of the state of the computation. This layer sits between the application and the OpenMP runtime layer, and intercepts all calls from the instrumented application program to the OpenMP library. This design permits us to implement the coordination protocol without modifying the underlying OpenMP implementation. This promotes modularity, eliminates the need for access to OpenMP library code, which is proprietary on some systems, and allows us to easily migrate from one OpenMP implementation to another. Furthermore, it is

relatively straightforward to combine our shared-memory checkpointer with existing application-level checkpointers for MPI programs to provide fault tolerance for hybrid MPI/OpenMP applications.

3.1 Tool Usage

C^3 can be used as a pass before an application's source code is run through the system's native compiler. The process of generating a fault tolerant application can be broken down into several steps. This process is easily automated and can be hidden inside a script, in much the same way that the details of linking with an MPI library are often hidden inside a mpicc script.

- Use the native preprocessor to translate the original source code into its corresponding pure C form. This involves applying `defines`, resolving `ifdefs` and inserting into the source code the files specified by `include` statements.
- The resulting preprocessed files are then given to C^3, which instruments them in a way that allows them to record their own state.
- The instrumented fault-tolerant files are fed to the native C compiler and linked to the C^3 coordination layer that keeps track of the application's interactions with OpenMP and coordinates the threads' checkpoints

In practice a user would use a single script to do all of the above actions, providing a list of files to be compiled and receiving a fault tolerant executable in return.

3.2 Protocol

We use a blocking protocol to co-ordinate the saving of state by the individual threads. This protocol has three phases, shown pictorially in Figure 2.

1. Each thread calls a barrier.
2. Each thread saves its private state. Thread 0 also saves the system's shared state.
3. Each thread calls a second barrier.

We assume that a barrier is a memory fence, which is typical among shared memory APIs. It is easy to see that if the application does not itself use synchronization operations such as barriers, its input-output behavior will not be changed by using this protocol to take checkpoints. The only effect of the protocol from the perspective of the application is to synchronize all threads and enforce a consistent view of the shared state by using a memory fence operation (normally implemented implicitly within the barrier). This state may not be identical to the system's state had a checkpoint not been taken. However, it is a legal state that the system could have entered since all consistency models only define the latest point at which a memory fence operation can take place, not the earliest (that is, it is always legal to include an additional memory fence operation). Furthermore, it is obvious that the state visible to each thread immediately after the checkpoint is identical to the state saved in the checkpoint.

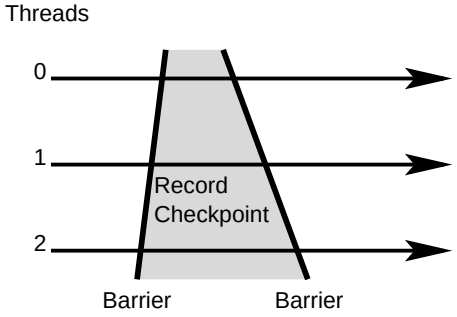


Figure 2: High-level view of checkpointing protocol

These properties ensure that we can restart the program by restoring all shared memory locations to their checkpointed values. Intuitively, if it was legal to flush all caches and set every thread's view of the shared memory to that memory image, then by restoring the entire shared address space to the image and flushing all the caches, we will return the system to an equivalent state.

The recovery algorithm follows from this, and is described below.

1. All threads restore their private variables to their checkpointed values and thread 0 restores all the shared addresses to their checkpointed values.
2. Every thread calls a barrier.
   This recovery barrier is necessary to make sure that the entire application state has been restored before any thread is allowed to access it.
3. Every thread continues execution.

Our protocol inserts additional barriers into the execution of the program and it is possible for these barriers to cross the application's own barriers and lock acquisitions. In such cases the checkpointing process may be corrupted or a deadlock may occur. To deal with this problem our protocol may force checkpoints to happen before the application's barriers and lock acquires, ensuring that no checkpoint conflicts with the application's causal interactions.

4. EXPERIMENTAL EVALUATION

Application-level checkpointing increases the running times of applications in two different ways. Even if no checkpoints are taken, the instrumented code executes more instructions than the original application to perform bookkeeping operations . Furthermore, if checkpoints are taken, writing the checkpoints to disk adds to the execution time of the program. In this section, we present experimental results that measure these two overheads for the C^3 system.

For our benchmark programs, we decided to use the codes from the SPLASH-2 suite [26] that we converted to run on OpenMP. We omitted the `cholesky` benchmark because it ran for only a few seconds, which was too short for accurate overhead measurement. We also omitted `volrend` because of licensing issues with the tiff library, and `fmm` because we could not get even the unmodified benchmark to run on our platforms.

One of the major strengths of application-level checkpointing is that the instrumented code is as portable as the original code. To demonstrate this, we ran the instrumented SPLASH-2 benchmarks on three different platforms: a 2-way Athlon machine running Linux, a 4-way Compaq Alphaserver running Tru64 UNIX, and an 8-way Unisys SMP system running Windows. In this section, we present overhead results on the first two platforms; we were not able to complete the experiments on the third platform in time for inclusion in this paper.

Benchmark	Problem size	Uninstrumented run time	C^3-instrumented run time 0 checkpoints taken	C^3-instrumentation overhead
fft	2^{24} data points	20s	20s	0%
lu-c	5000×5000 matrix	110s	110s	0%
radix	100,000,000 keys, radix=512	30s	31s	3%
barnes	16384 bodies, 15 steps	103s	106s	3%
ocean-c	514×514 ocean, 600 steps	162s	162s	0%
radiosity	Large Room	8s	8s	0%
raytrace	Car Model, 64MB RAM	32s	34s	6%
water-nsquared	4096 molecules, 60 steps	260s	223s	-14%
water-spatial	4096 molecules, 60 steps	156s	141s	-9%

Table 1: SPLASH-2 Linux Experiments

4.1 Linux/x86 Experiments

The Linux experiments were conducted on a 2-way 1.733GHz Athlon SMP with 1GB of RAM. The operating system was SUSE 8.0 with a 2.4.20 kernel. The applications were compiled with the Intel C++ Compiler Version 7.1. All experiments were run using both processors (i.e. P=2). Checkpoints were recorded to the local disk. The key parameters of the benchmarks used in the Linux experiments are shown in Table 1.

4.1.1 Execution Time Overhead

In this experiment, we measured the running times of (i) the original codes, and (ii) the instrumented codes without checkpointing. Times were measured using the Unix `time` command. Each experiment was repeated five times, and the average is reported in Table 1. From the spread of these running times, we estimate that the noise in these measurements is roughly 2-3%. The table shows that for most codes, the overhead introduced by C^3 was within this noise margin. For two applications, `water-nsquared` and `water-spatial`, the instrumented codes ran faster than the original, unmodified applications. Further experimentation showed that this unexpected improvement arose largely from the superior performance of our heap implementation compared to the native heap implementation on this system. We concluded that the overhead of C^3 instrumentation code for the SPLASH-2 benchmarks on the Linux platform is small, and that it is dominated by other effects such as the quality of the heap implementation.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Recovery
fft	765	43	22
lu-c	191	2	5
radix	768	43	24
barnes	569	4	10
ocean-c	56	1	4
radiosity	32	0	1
raytrace	68	0	2
water-nsquared	4	1	0
water-spatial	3	0	0

Table 2: Overhead of Checkpoint and Recovery on Linux.

4.1.2 Checkpoint and Recovery Overhead

Finally, we measured the execution time overhead of taking a single checkpoint and performing a single recovery. These numbers can be used in formulas containing particular checkpointing frequencies and hardware failure probabilities to derive the overheads for a long-running application.

To measure the overhead of taking a single checkpoint, we ran the C^3-transformed version of each benchmark without taking a checkpoint and compared its execution time to the time it took to run the same benchmark and taking a single checkpoint.

To measure the overhead of a single recovery, we first measure the time of execution from the start of the program until after the single checkpoint completes. Then we add to this the time measured from the beginning of a restart from this checkpoint to the end of the program. Finally, from this sum, we subtract the execution time for the complete program that takes a single checkpoint.

The results are shown in Table 2. The time to take checkpoints is fairly low for most applications, and is significant only for applications for which checkpoint sizes are very large (`fft` and `radix`). As mentioned before, these checkpoints were saved to local disk on the machine. If they were saved to a networked file system, we would expect the overheads to be larger.

4.2 Alpha/Tru64 Experiments

The Alpha experiments were conducted at the Pittsburgh Supercomputing Center on the Lemieux cluster. This cluster is composed of 750 Compaq Alphaserver ES45 nodes. Each node is an SMP with 4 1Ghz EV68 processors and 4GB of memory. The operating system is Compaq Tru64 UNIX V5.1A. All codes were run on all 4 processors of a single node (i.e. P=4). Checkpoints were recorded to system scratch space, which is a networked file system available from all nodes. The key parameters of the SPLASH-2 benchmarks used in the Alpha experiments are shown in Table 3.

4.2.1 Execution Time Overhead

We measured the overheads of instrumentation on Lemieux using the same methodology we used for Linux. Table 3 shows the results.

These results show that except for `radix` and `ocean-c`, the overheads due to C^3's transformations are either negligible or negative. The overheads in `radix` and `ocean-c` arise from two different problems that we are currently addressing.

The overhead in `radix` comes from some of the details of how C^3 performs its transformations. Our state-saving mechanism computes addresses of all local and global variables, which may prevent the compiler from allocating these variables to a register. For `radix`, it appears that this inability to register-allocate certain variables leads to a noticeable loss of performance. We are currently re-designing the mechanism to circumvent this problem.

Our experiments also showed that the overhead in `ocean-c` execution comes from our heap implementation

Benchmark	Problem size	Uninstrumented run time	C^3-instrumented run time 0 checkpoints taken	C^3-instrumentation overhead
fft	2^{26} data points	68s	67s	-2%
lu-c	12000×12000 matrix	719s	724s	1%
radix	300,000,000 keys, radix=512	61s	70s	15%
ocean-c	1026×ocean, 600 steps	153s	183s	20%
radiosity	Large Room	13s	12s	-9%
raytrace	Car Model, 1GB RAM	20s	20.4s	2%
water-nsquared	12167 molecules, 10 steps	136s	140s	3%
water-spatial	17576 molecules, 40 steps	214s	218s	2%

Table 3: Characteristics and Results of SPLASH-2 Alpha Experiments

(replacing our heap implementation with the native heap eliminated this overhead). While this implementation has been optimized for Linux, it is not as optimized for Alpha. This tuning is underway.

4.2.2 Checkpoint and Recovery Overhead

Table 4 shows the checkpoint time and the recovery time for the different applications. It can be seen that there is a correlation between the sizes of the checkpoints and the amount of time it takes to perform the checkpoint. In these experiments, the checkpoint files were written to the system scratch space rather than to a local disk, so for codes that take larger checkpoints, the overheads observed on Lemieux are higher than the overheads on the Linux system shown in Table 2.

Benchmark	Checkpoint Size (MB)	Seconds per Checkpoint	Seconds per Recovery
fft	3074	363	32
lu-c	1103	136	7
radix	2294	285	36
ocean-c	224	68	*
radiosity	43	8	1
raytrace	1033	137	7
water-nsquared	16	3.75	388
water-spatial	12	3.5	17

Table 4: Overhead of each checkpoint and recovery on Alpha.

The only code with a high recovery overhead is `water-nsquared`, and it highlighted an inefficiency in our current implementation. Note that `water-nsquared` takes 3.5 seconds to record a 16MB checkpoint but takes 388 seconds to recover. The reason for this is that `water-nsquared malloc()-s` a large number of individual objects: 194K. This in comparison to the 18K objects that `water-spatial` allocates or the 65K allocated by `water-nsquared` given the input parameters used on Linux. C^3's checkpointing code is optimized to use buffering when writing these objects to a checkpoint, but its recovery code does not have such optimizations, so it performs one file read for every one of these objects. The cost of that many file reads, even to buffered files is very high and results in a long recovery time. Our next implementation of the C^3 system will optimize reading the checkpoint files to eliminate this inefficiency.

`Ocean-c`'s recovery overhead was measured to be negative. However this negative overhead was within the variability of the timing results in this experiment, so it appears to be an artifact of the fluctuations inherent to a networked file system.

4.3 Discussion

When we began this work, we invested considerable time in refining our coordination protocol because we thought that the execution of the protocol would increase the running time of the application significantly. Indeed, much of the literature on fault-tolerance focuses on protocol optimizations such as reducing the number of messages required to implement a given protocol.

Our experiments showed that the overheads are largely due to other factors, summarized below.

- The performance of some codes is very sensitive to the memory allocator. Overall, we obtained good results on the Linux system because we have tuned our allocator for this system; on Lemieux, where the tuning work is still ongoing, some codes such as `ocean-c` had higher overheads.
- The instrumentation of code to enable state-saving prevents register allocation of some variables in codes like `radix` on Lemieux. This is relatively easy to fix by introducing new temporaries, and it is being implemented in our preprocessor.
- For codes that produce large checkpoint files, the time to write out these files dominates the checkpoint time. We are exploring incremental checkpointing, as well as compiler analysis, to reduce the amount of saved state.
- Finally, recovery time for codes that create a lot of small objects, such as `water-nsquared` on Lemieux, needs to be reduced by better management of file I/O.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented an implementation of a blocking, coordinated checkpointing protocol for application-level checkpointing (ALC) of shared-memory programs using locks and barriers. The implementation has two components: (i) a pre-compiler that automatically instruments C/OpenMP programs so that they become self-checkpointing and self-restarting, and (ii) a runtime layer that implements the co-ordination protocol. Experiments with SPLASH-2 benchmarks show that the overheads introduced by our implementation are small. The implementation can be used to checkpoint shared-memory programs; it can also be used in concert with a system for checkpointing message-passing programs, such as [5, 4, 24], to provide a solution for checkpointing hybrid message-passing/shared-memory programs.

Our ALC approach has the advantage that programs instrumented by our pre-compiler become self-checkpointing and self-restarting, so they become fault-tolerant in a

platform-independent manner. This is a major advantage over system-level checkpointing approaches, which are very sensitive to the architecture and operating-system. We have demonstrated this platform-independence by running on a variety of platforms.

In the future, we intend to extend (C^3) to deal with a broader set of shared-memory constructs. In particular, we intend to support the full OpenMP standard. Furthermore, we intend to couple (C^3) with the MPI checkpointer described in [4] to produce a fault tolerance solution for programs using both message-passing and shared-memory constructs.

6. REFERENCES

[1] A. Beguelin, E. Seligman and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1995.

[3] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997. Also available as `http://citeseer.nj.nec.com/beguelin97application.html`.

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *Proceedings of the 2003 International Conference on Supercomputing*, pages 234–243, June 2003.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 84–94, June 2003.

[6] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *IEEE Transactions on Computing Systems*, 3(1):63–75, 1985.

[7] Condor. http://www.cs.wisc.edu/condor/manual.

[8] W. Dieter and Jr. J. Lumpp. A user-level checkpointing library for POSIX threads programs. In *Proceedings of 1999 Symposium on Fault-Tolerant Computing Systems (FTCS)*, June 1999.

[9] J. Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. http://www.nersc.gov/research/FTG/checkpoint/reports.html.

[10] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[11] P. Guedes and M. Castro. Distributed shared object memory. In *Proceedings of WWOS*, 1993.

[12] T. Tannenbaum J. B. M. Litzkow and M. Livny. Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System. Technical Report Technical Report 1346, University of Wisconsin-Madison, 1997.

[13] Angkul Kongmunvattan, S. Tanchatchawal, and N. Tzeng. Coherence-based coordinated checkpointing for software distributed shared memory systems. In *Proceedings of the International Conference on Distributed Computer Systems (ICDCS 2000)*, 2000.

[14] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[15] J.S. Plank M. Beck and G. Kingsley. Compiler-Assisted Checkpointing. Technical Report Technical Report CS-94-269, University of Tennessee, December 1994.

[16] Y. M. Wang M. Elnozahy, L. Alvisi and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report Technical Report CMU-CS-96-181, Carnegie Mellon University, October 1996.

[17] Z. Zhang M. Prvulovic and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared memory multiprocessors. In *International Conference on Computer Architecture*, 2002.

[18] K. Kusano M. Sato, S. Satoh and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, September 1999.

[19] Message Passing Interface Forum (MPIF). MPI: A message-passing interface standard. Technical Report, University of Tennessee, Knoxville, June 1995.

[20] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, C. Vizino. A checkpoint and recovery system for the pittsburgh supercomputing center terascale computing system. http://www.psc.edu/publications/tech_reports/chkpt_rcvry/checkpoint-recovery-1.0.html.

[21] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Symposium on Principles of Distributed Computing Systems (PDCS)*, 1994.

[22] OpenMP Architecture Review Board. *OpenMP C and C++ Application, Program Interface*, Version 1.0, Document Number 004–2229–01 edition, October 1998. Available from http://www.openmp.org/.

[23] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA 2002)*, July 2002.

[24] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of International Parallel Processing Symposium(IPPS)*, 1996.

[25] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996. Also available at `http://citeseer.nj.nec.com/stellner96cocheck.html`.

[26] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture 1995*, pages 24–36, June 1995.

Nanos Mercurium: a Research Compiler for OpenMP

J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé and J. Labarta
Computer Architecture Department, Technical University of Catalonia,
cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain
{jbalart, aduran, marc, xavim, eduard, jesus}@ac.upc.es

Abstract

OpenMP is still in the process of being defined and extended to broad the range of applications and parallelization strategies it can be used for. The proposal of OpenMP extensions may require the implementation of new features in the runtime system supporting the OpenMP parallel execution and modifications in an existing OpenMP compiler, either at the front end (parsing of new directives and clauses) or back end (code generation targetting a specific runtime system). It may even imply to modify the internals of the compiler to accommodate new concepts, data structures or code manipulation routines. The objective of the Nanos Mercurium compiler is to offer a compilation platform that OpenMP researchers can use to test new language features. It needs to be: 1) robust enough to enable testing with real applications and benchmarks, and 2) easy to add new features into it, hiding as much as possible all its internals. To achieve the first objective, we decided to build Nanos Mercurium on top of an existing compilation platform, the Open64 compiler. In order to achieve the second objective, Nanos Mercurium uses templates of code for specifying the transformations of the OpenMP directives. In order to validate Nanos Mercurium, we show how to implement dynamic sections, a relaxation of the current definition of SECTIONS to allow the parallelization of programs that use iterative structures (such as while loops) or recursion. It is shown as an alternative to the workqueueing execution model proposed by KAI/Intel.

1 Introduction

Shared-memory parallel architectures are becoming common platforms for the development of CPU demanding applications. Today, these architectures are found in the form of small symmetric multiprocessors on a chip, on a board, or on a rack with a modest number of processors (usually less than 32 or 64). In order to benefit from the potential parallelism they offer, programmers require programming models to develop their parallel applications with a reasonable performance/simplicity trade–off. These programming models are usually offered as library implementations or extensions to sequential languages that allow the programmer to express the available parallelism in the application. Language extensions are defined by means of directives and language constructs that are translated by the compiler to library calls. These libraries offer mechanisms to create threads, distribute work among them and synchronize their activity.

OpenMP [5] has become the standard for parallel programming in shared memory parallel architectures. OpenMP offers a set of directives or pragmas that are included in the specification of high-level languages such as C or Fortran. OpenMP results successful for the parallelization of a broad range of numerical applications coming from different engineering fields. Basically, the success comes from the fact that OpenMP makes the parallelization process easy and incremental.

However, the OpenMP community is conscious about the fact that certain parallelization structures are not possible (or very difficult to implement) with the current definition. For example, the parallelization of non–numerical applications, which usually deal with memory linked data structures and/or rely on divide and conquer strategies. Those applications are coded through the use of iterative structures like `while` loops and recursion. For this reason, the language specification is open to extensions. The OpenMP Architecture Review Board is in charge of studying and discussing proposals that come from the OpenMP community. Application developers as well as researchers in compiler optimizations can submit their proposals for possible extensions of the specification. For example, KAI/Intel proposed workqueueing to handle the parallelization of the above mentioned non–numerical codes [9, 10]. The proposal is based on the introduction of a new work–sharing construct, which specifies a queue (`TASKQ`) where tasks are queued for execution (`TASK`). Threads that belong to the team dynamically extract tasks from that queue.

The process of designing, implementing and evaluating

a new proposal could be more effective if there were open source compiler and runtime infrastructure for OpenMP simple to modify and extend. There have been several recent attempts, as for instance NanosCompiler [1] for Fortran77, Omni [7] for Fortran77 and C and OdinMP [4] for C/C++. All of them are source–to–source translators that transform the code into an equivalent version with calls to the associated runtime system. However, modifying the compiler to include new language features, to explore alternative code generation strategies or to target different runtime systems is not an easy task that requires to go into the deep internals of the compiler. Nanos Mercurium is presented as a new approach to build an infrastructure for OpenMP extensions. It has been designed with two objectives: 1) it needs to be robust enough to enable testing with real applications and benchmarks, and 2) it has be easy to add new features into it, hiding as much as possible all its internals. To achieve these objectives we decided to build Nanos Mercurium on top of an existing compilation platform, the Open64 compiler, and to use templates to specify the transformations for OpenMP construct.

The paper is structured as follows: Section 2 describes the basic OpenMP compiler transformations. Section 3 describes the approach followed in the design of Nanos Mercurium: specification of code transformations based on templates. As an example, Section 4 presents a relaxation of the OpenMP `SECTIONS` work–sharing construct and its implementation in Nanos Mercurium. Section 5 evaluates the performance achieved with some codes and compares this performance with the one achieved with the Intel compiler with taskqueues. Finally, Section 6 concludes the paper and outlines future work.

2 OpenMP Compiler Support

In this section we summarize the common functionalities available in most runtime systems that support the OpenMP execution model, and show the compiler transformations needed in the source-to-source translation process. The most important features need to be considered: how threads are created/terminated, how threads execute the code inside the parallel region, how threads parcel out the work specified in work–sharing constructs, how threads access data according to the data scoping rules and the mechanisms available to synchronize threads at specific points. All these features are jointly achieved by the runtime and compiler. In this paper, we show the transformations one by the Nanos Mercurium compiler when targeting the NthLib OpenMP runtime. We consider the description of the NthLib internals out of the scope of this paper (for more details the reader is referred to [3, 2]). Both compiler and runtime are available from the project web site at CEPBA (www.cepba.upc.es/mercurium).

```
void foo (double A [100][100],double B [100][100],)
{
#pragma omp parallel
 {
#pragma omp sections
#pragma omp section
  {
   initialize (A );
  }
#pragma omp section
  {
   initialize (B );
  }
 }
}
```

a) Example code with SECTIONS construct

```
void par_foo_01 (double A [100][100],double B [100][100] )
{
 call sections_foo_01 (A ,B )
}
```

b) Parallel code encapsulation

```
void foo (double A [100][100],double B [100][100],)
{
 nth_nthreads = nth_cpus_actual ():
 nth_depadd (nth_self (),nth_nthreads + 1 );
 for (th = 0;th < nth_nthreads;th++ )
  nth_thread_create_1s_vp (par_foo_01,0,th,2,A ,B )
 nth_block ();
}
```

c) Code for thread creation/termination

```
void sections_foo_01 (double A [100][100],double B [100][100],)
{
 nth_begin_for (0,1,1,DYNAMIC,1 );
 while (nth_next_items (&nth_down,&nth_up,nth_last)) {
  for (nth_sect = nth_down;nth_sect < nth_up;nth_sect) {
   if (nth_sect = = 0 ) section_foo_00 (A );
   if (nth_sect = = 1 ) section_foo_01 (B );
  }
 }
 nth_end_for (1 );
}
```

d) Code for SECTIONS work sharing construct

Figure 1. OpenMP example and compiler transformations.

The transformation process is described using the sample code shown in Figure 1.a. This section will mainly focuses on code transformations for thread creation and termination, and for work distribution. The reader is referred to [8] for a description of the other compiler transformations.

2.1 Parallel Code Encapsulation

In order to allow threads to execute the code encapsulated in the `PARALLEL` construct, the compiler extracts the code enclosed in a parallel region and defines a subroutine where the code is located. This transformation is quite simple and just requires basic compiler operations like code extraction and symbol reference gathering. This last one is necessary to declare subroutine arguments that define the correct context to execute the encapsulated code. For the example in 1.a, the compiler generates the subroutine in figure

1.b.

2.2 Thread Spawn and Join

Once the parallel code has been encapsulated, the compiler replaces the original code by a block of statements in charge of spawning and joining the team of threads created to execute it. This block of statements mainly contains a loop that adds a new thread to the team in each iteration. The compiler injects a runtime call to the `nth_create_1s_vp` service to create the thread, provide the address of the encapsulated function to execute and the necessary arguments. Before the execution of this loop, the compiler injects code to query the runtime about the number of available threads to create the team. This corresponds to the runtime invocation to `nth_cpus_actual` service. Once all threads are created, the thread that has spawned the parallelism blocks and waits until all threads in the team finish the execution of the parallel code. This is accomplished by the injection of another runtime call to service `nth_block` that implements the implicit barrier at `END PARALLEL`. Figure 1.c shows the code that the compiler should generate for the parallel execution of the example in Figure 1.a.

2.2.1 Work Distribution

For each work-sharing construct in a parallel region, the compiler performs a transformation process similar to the one described for the parallel region. First the code inside the work-sharing construct is extracted and encapsulated in a new subroutine. The compiler injects a call statement to the new allocated subroutine that replaces the extracted code. Notice that this is done in the subroutine containing all the code in the parallel region, as this transformation is done after the code encapsulation process described in previous section. The code inside the new encapsulated function depends on the work-sharing (i.e. `DO`, `SECTIONS`, `SINGLE` or `WORKSHARE`). For example, for a `SECTIONS` construct, the compiler would generate the code shown in Figure 1.d. The runtime services `nth_begin_for` and `nth_end_for` define and conclude the execution of the work-sharing construct. The compiler generates a loop that each thread in the team executes to request work to execute (service `nth_next_iters`). The body of this loop simply redirects the execution to the appropriate `SECTION` according to their lexicographic order in the source code. The reader is referred to [8] for additional details about this code and code generation for other work–sharing constructs.

2.3 Parametrization of the Transformations

All the compiler transformations at this point have aspects that depend on the source code that is being transformed while other aspects are independent. For example, the symbol context depends strictly on the symbols that are referenced inside the parallel code and the OpenMP data scoping clauses. Or for example, the injection of the call to `nth_create_1s_vp` is always done in the same way except for some of its arguments. For this reason, the proposed mechanism in this paper to specify the transformation process allows the specification of constant and variable parts, i.e. parts that depend or not on the source code being transformed.

3 Template–guided Transformations

The Nanos Mercurium compiler is based on the definition of a set of templates that guide the compiler in the process of transforming the code according to the OpenMP directives. In order to have a robust platform able to handle real applications and benchmarks, we decided to implement Nanos Mercurium on top of the Open64 compiler [6]. We started from Open64 with a built–in OpenMP parser and internal representation, named `whirl`, able to represent OpenMP constructs. In order to drive the transformation process, we added a new phase that is executed after the corresponding Fortran or C front–end. The input of this new phase is the `whirl` intermediate representation created by the front–end with the OpenMP directives represented in the Abstract Syntax Tree (AST). When our phase is executed, all the directives are transformed applying a set of templates, obtaining a new `whirl` representation with all the OpenMP directives replaced by calls to the NthLib thread library. After that, we use one of the Open64 back–ends over the `whirl` intermediate representation that emits source code in the same input language.

3.1 Template Definition

The templates used by our compiler are written in the same language as the application being compiled, with no additional extensions to it. There are three simple rules that are needed to write templates. The first rule describes how to specify a variable part in a template, the second one how to specify parts of code only necessary under certain conditions and the last one how to specify parts of code that have to be repeated a variable number of times.

- `Rule 1`: All variables in a template whose names start with a certain prefix, `tpl_` in our implementation, are template variable parts that the compiler will replace with an expression. The transformation to do over this variables is coded in the compiler internals. The compiler includes a correspondence between template variable names and the corresponding transformation.

- `Rule 2`: When the logical expression in a conditional construction (if statement) includes a template variable, only the code that would be executed after evaluating the condition will be inserted in the template instantiation. The template expansion module has to evaluate the condition depending on the code being compiled.

- `Rule 3`: When the logical expression of a loop construction (`dowhile` statement) includes a template variable, the code inside the construction is inserted `n` times and the original loop construction is extracted from the template instantiation. The template expansion module has to evaluate the condition and the value `n`, depending on the code being compiled.

Rule 1 needs to be extended in order to handle the possibility of having a variable outside any expression or language construction. In these cases we declare external subroutines with names according to rule 1 and insert calls to them in the templates. These subroutine calls will be replaced by the expansion of the template reference by the subroutine name. In order to support the previously defined rules, our implementation has required a template expansion module that performs the following template operations:

- `Operation 1`: replace a template variable by another variable.
- `Operation 2`: replace a template variable by a integer or real constant.
- `Operation 3`: replace a template variable by a portion of the AST.
- `Operation 4`: replace a template variable by a set of real parameters.
- `Operation 5`: replace a template variable by a set of formal parameters.
- `Operation 6`: create a set of local variables in a template.
- `Operation 7`: replicate part of code in template.
- `Operation 8`: extract a part of code in a template.
- `Operation 9`: change the name of a template variable.

The current status of the Nanos Mercurium compiler supports near the whole language specification (OpenMP 2.0). All the compiler transformations are specified through the use of code templates. For each language construct there is one template code to be applied. In [8] we presented how the code templates are used to implement the necessary support in the compiler for `PARALLEL` and work–sharing constructs. In the following section, we show how to build a new template to instruct the compiler to transform a relaxed version of the `SECTIONS` construct, what we call dynamic sections.

4 Dynamic Sections Proposal

The proposal in this section tries to address the limitations in the OpenMP programming model regarding applications based on recursion and/or traversing memory linked data structures. The proposal is based on relaxing the current specification for `SECTIONS`, mainly allowing a `SECTION` to be instantiated multiple times inside the scope of `SECTIONS` and executing code outside any `SECTION` by a single thread. The proposal has certain similarities with the KAI/Intel workqueueing proposal [9, 10], hiding the concept of queues to the programmer. This proposal is used as an example to show how to extend Nanos Mercurium by defining the appropriate template to handle dynamic sections.

4.1 Relaxing the SECTIONS construct

The `SECTIONS` work-sharing construct allows the programmer to define several portions of code to be executed in parallel. OpenMP does not allow to have code outside any `SECTION` (in fact in the Fortran specification the first `SECTION` can be omitted but the compiler assumes that it exists). We relax this constraint so that this code outside all `SECTION` is executed single threaded. This thread would be the responsible for work generation, allowing the programmer to implement the work generation strategies that are commonly used in the applications mentioned before. In addition, a `SECTION` could now be instantiated multiple times, for example because it is included in a loop that traverses a linked–data structure or because it is inside a recursive code structure.

When a team of threads executing a `PARALLEL` region encounters a `SECTIONS` construct, only one of them is going to execute the code inside the `SECTIONS` construct and outside any `SECTION` construct. When the thread executing the single–threaded part of `SECTIONS` finds an instance of a `SECTION`, it simply queues the `SECTION` in an internal queue of work to be executed by the team of threads.

The code in figure 2 shows an example of use. Subroutine `queens_par` contains the definition of a parallel region. Inside this region, there is a `SECTIONS` construct that includes a `for` loop. In each iteration of this loop a `SECTION` construct is instantiated creating a new branch in the recursion tree.

One of the main differences with the workqueueing model is that this proposal does not allow the nesting of `SECTIONS` while KAI/Intel allow the nesting of `TASKQ`. The nesting should be done by nesting `PARALLEL` regions, each including the `SECTIONS`. The first implication of this is that the same team of threads is executing the nested `TASKQ`, while in our proposal a new team is created in each `PARALLEL` region, disabling the possibility of doing work

```
void queens_par (int board, int level, ...)
{
 int copy_board[MAX_SIZE];
 if (level == MAX_SIZE) return;
#pragma omp parallel
#pragma omp sections
 {
  for (i= 0;i< MAX_SIZE;i++) {
    #pragma omp section
     {
     for (j= 0;j<level;j++)
        copy_board[j] = board[j];
       ...;
     queens_par (copy_board, level+1,...);
     }
  }
 }
}
```

Figure 2. Dynamic sections example: queens code.

stealing. The second implication is that the implicit barrier at the end of PARALLEL does not exist when simply nesting TASKQ. We are investigating the effects that these implications may have in applications and see if they limit the parallelism that can be exploited. In general, it should be considered the possibility of nesting any kind of OpenMP work–sharing construct, but this is considered out of scope of this paper and subject of further discussion.

If the compiler detects that there is no code outside any SECTION, it can turn to the initial code generation strategy (what we name static sections).

4.2 Template Specification for Dynamic Sections

Figure 3 shows the template specification that instructs the compiler how to transform the code for a dynamic SECTIONS work-sharing construct. The code uses the support provided by the NthLib thread runtime, but the mechanisms to implement the compiler transformation are valid independently of the target runtime library. The template is written in C and includes all the necessary declarations to be a correct C code. Some of these declarations correspond to template variables which are going to cause the compiler apply the appropriate transformations.

The name of the subroutine where the template code is located is identified by a template variable TPL_DYNAMIC_SECTIONS which is going to cause the compiler to generate a new identifier for the subroutine. The template variable TPL_SECTIONS_PARAMS appears as a subroutine argument. It is going to be replaced by the declaration of all the symbols referenced by all the dynamic sections. The template code includes an `if` statement that checks if the thread is the master of the current team. This ensures that only the master thread is going to execute the code related to the work distribution. This code consists of several statements that cause the executing thread communicates to the runtime the number of dynamic sections that are going to be generated. This number is obtained by the compiler after the transformation applied to the template variable TPL_NUM_SECTIONS is done. This variable expands to the number of sections in the SECTIONS work-sharing construct. The runtime call nth_depadd communicates this number to the runtime. After that, a `do while` loop is coded in the template, but controlled by a template variable: TPL_SECTIONS. This is going to cause the compiler traverses the list of sections. For each section, the code in the loop body is replicated and the appropriate transformations are applied in it. The loop body in the template is formed by several statements where the TPL_SECTION_NUM_PARAMS, TPL_SECTION_SUBROUTINE, TPL_SECTION_PARAMS variables appear. The compiler replaces the variable TPL_SECTION_NUM_PARAMS by the number of arguments in the section's subroutine. The TPL_SECTION_SUBROUTINE variable appears as argument of the runtime call to service nth_create_1s. This variable is replaced by the identifier of the routine generated for the section being treated. Finally, the variable TPL_SECTION_PARAMS is replaced by the list of arguments that are referenced inside the section's code.

The OpenMP definition forces the threads to perform a barrier synchronization at the end of the SECTIONS execution. Notice that the SECTIONS construct is the only work-sharing inside the parallel region. This allows for optimization and removing the barrier as it is going to be performed at the END PARALLEL construct. This is achieved in the template code by the runtime calls to nth_successor and nth_depadd. Both calls are responsible to link the dynamic sections with the thread that spawned the parallel region, forcing that this one can not be terminated until all the dynamic sections are terminated too.

Notice that most of the variable parts in the template correspond to information that the compiler has to compute, no matter the code generation associated to the dynamic sections. If this code generation has to be changed, the changes can be done in the template, with no need of implementing the changes inside the internals of the compiler.

```
void TPL_DYNAMIC_SECTIONS(int TPL_SECTIONS_PARMS)
{
 long long nth_succ;
 int nth_nsections;
 int ndep;
 int nth_num_params;
 long long nth;
 int TPL_NUM_SECTIONS;
 bool TPL_SECTIONS;
 int TPL_SECTION_NUM_PARMS;
 int TPL_SECTION_PARMS;

 if (nth_iam_master_())
 {
  nth_succ = nthf_successor(nthf_self_());
  nth_sections = TPL_NUM_SECTIONS + 1;
  nthf_depadd_(&nth_succ, &nth_nsections);
  nth_ndep = 0;
  while (TPL_SECTIONS)
  {
    nth_num_params = TPL_SECTION_NUM_PARMS;
    nth = nthf_create_1s_(TPL_SECTION_SUBROUTINE,
                          &ndep,
                          &nth_succ,
                          &nth_num_params,
                          TPL_SECTION_PARMS);
     nth_to_rq(nth);
  }
 }
}
```

Figure 3. Template for dynamic sections.

5 Testing and Evaluation

In this section we describe the tests performed to validate the proposed dynamic sections and the transformation done by the compiler to implement them.

5.1 Benchmarks

In order to test our proposal and implementation, we have used some of the examples that KAI/Intel used to motivate and evaluate their implementation of taskqueuing. We mainly replaced `TASKQ` with `SECTIONS` and `TASK` with `SECTION`. All of them have in common that the algorithms used in the computations are expressed through recursion. Parallelism appears among the recursion branches that each application defines:

- Queens: The algorithm computes the different solutions for placing `n` chess queens in a chess board, none of them "killing" another. The computation is done recursively. Each recursive branch in the recursion tree, can be executed in parallel.
- Strassen: Strassen matrix multiplication algorithm decomposes square matrices into four square sub-matrices and then compute the whole matrix multiplication performing only seven sub-matrices multiplications and eighteen sub-matrices additions/substractions. Recursive decomposition can be applied to create as many matrix multiplication tasks as desired and execute all them in parallel.
- MultiSort: Multisort is a variation of the mergesort algorithm. A recursive divide and conquer approach is used. An array of N 64-bits elements is divided in four sub-arrays, each sub-array is sorted recursively and finally the four sorted sub-arrays are merged first to obtain two sorted parts and lastly only one.

5.2 Initial Experiments

The experimental platform for this paper is a SGI Origin2000 with 64 R10000 processors (250 Mhz) and Irix 6.5. Nanos Mercurium generates a transformed source code that is finally compiled and linked with NthLib using the SGI native compiler to generate the executable code.

Figure 4 shows the performance, in terms of speed–up with respect to the sequential version, that is obtained for the benchmarks. These numbers are initial and there is still more work to do in order to understand them and detect who is the responsible for the performance degradation observed (application, dynamic sections proposal, compiler transformation or OpenMP runtime):

- `Queens`: the benchmark is executed with a chess board of 13 x 13. Each time the parallelism is spawn, 13 dynamic sections are created. In order to feed 4, 8, 16 , and 32 threads it suffices to have 2 levels of recursion. As shown in Figure 4 the application scales quite well with the number of processors.
- `Strassen`: the benchmark is executed with a matrix of 1280 doubles. As shown in Figure 4 the benchmark achieves efficiency up to 16 threads. Using more threads is not providing more parallelism (speedup of 16.95 with 32 threads).
- `MultiSort`: This is the application that behaves worst. Although the speed–up increases up to 16 processors, the efficiency decreases when more processors are used (0.72 with 2 processors and 0.27 with 16).

6 Conclusions and Future Work

In this paper we present Nanos Mercurium, a new research compiler infrastucture that we want to offer to the OpenMP community to test new proposals to the language. The compiler source–to–source transformations are specified using templates, which relieves the user from going into the deep internals of the compiler.

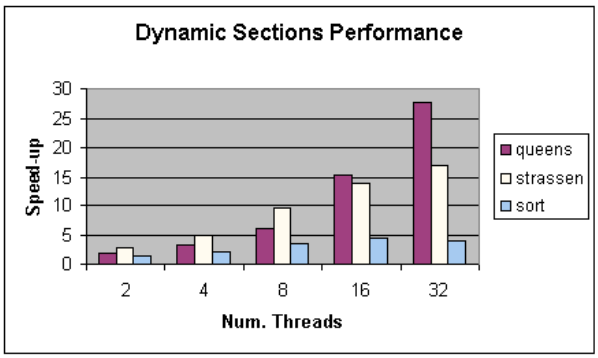


Figure 4. Performance for the Nanos Mercurium implementation of dynamic sections.

The templates for the main OpenMP constructs are described in a companion paper [8]. In this paper we want to show how to specify the template for a relaxed definition of `SECTIONS`, what we call dynamic sections. Dynamic sections allow to extract parallelism out of programs that make use of recursion or iterative traversals of data structures to generate work to be executed in parallel. The proposal and implementation are tested using some benchmarks and their performance is evaluated.

Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain under contract TIC2001-0995-C02-01 and the European Union under contract IST-2001-33071.

References

[1] M. Gonzalez, E. Ayguade, J. Labarta, X. Martorell, N. Navarro and J. Oliver. ”NanosCompiler: A Research Platform for OpenMP Extensions”, In First European Workshop on OpenMP, Lund (Sweden), October 1999

[2] X. Martorell, E. Ayguad, N. Navarro, J. Corbalan, M. Gonzlez and J. Labarta, ”Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors”, Proceedings of the 13th. ACM International Conference on Supercomputing (ICS99).

[3] X. Martorell, E. Ayguad, N. Navarro and J. Labarta, ”A Library Implementation of the Nano-Threads Programming Model”, Proceedings of the 2nd. Europar Conference 1996.

[4] C. Brunschen and M. Brorsson, Lund University ”OdinMP/CCp – A Portable Implementation of OpenMP for C”, First European Workshop on OpenMP, (EWOMP99)

[5] ”Fortran Language Specification , v2.0”, http:/www.openmp.org

[6] http://sourceforge.net/projects/open64/

[7] http://phase.hpcc.jp/Omni/

[8] J. Balart, A. Duran, M. Gonzalez, X. Martorell, E. Ayguade, J. Labarta, ”Skeleton driven compiler transformations”, 11th Workshop on Compilers for Parallel Computers, July 7-9

[9] S. Shah, G. Haab, P. Petersen, J. Throop, ”Flexible Control Structures for Parallel C/C++”, First European Workshop on OpenMP, September 1999, Lund University, Lund, Sweden

[10] E. Su, X. Tian, M. Girkar, H. Grant, S. Shah, P. Peterson, ”Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures” Fourth European Workshop on OpenMP, September 2002, Rome