Towards OpenMP Execution on Software Distributed Shared Memory Systems

Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann *

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
http://www.ece.purdue.edu/ParaMount

Abstract. In this paper, we examine some of the challenges present in providing support for OpenMP applications on a Software Distributed Shared Memory(DSM) based cluster system. We present detailed measurements of the performance characteristics of realistic OpenMP applications from the SPEC OMP2001 benchmarks. Based on these measurements, we discuss application and system characteristics that impede the efficient execution of these programs on a Software DSM system. We point out pitfalls of a naive translation approach from OpenMP into the API provided by a Software DSM system, and we discuss a set of possible program optimization techniques.

1 Introduction

The current OpenMP 2.0 API is designed to be used on shared-memory multiprocessors. In this paper we describe challenges and opportunities in providing OpenMP support via a Software Distributed Shared Memory (DSM) system. Implementing OpenMP via Software DSM [1] is one possible avenue for making OpenMP amenable to distributed-memory systems, such as cluster architectures.

Alternative approaches to OpenMP implementation on distributed systems include language extensions and architecture support. Several recent papers have proposed language extensions. For example, in [2–4] the authors propose data distribution directives similar to the ones designed for High-Performance Fortran (HPF). Other researchers have proposed page placement techniques to map data to the most suitable processing nodes [2]. In [5] remote procedure calls are used to employ distributed processing nodes. Another related approach is to use explicit message passing for communication across distributed systems and OpenMP within shared-memory multiprocessor nodes [6]. Providing architectural support for OpenMP essentially means building shared-memory multiprocessors (SMPs). While this is not new, an important observation is that increasingly large-scale SMPs continue to become commercially available. For example, the largest machine on which SPEC OMP 2001 benchmarks have

* This work was supported in part by the National Science Foundation under Grant No. 9703180, 9975275, 9986020, and 9974976.

been reported recently is a 128-processor Fujitsu PRIMEPOWER2000 system (www.spec.org/hpg/omp2001/results/omp2001.html).

The present paper is meant to complement, rather than compete with, these related approaches. The goal is to measure quantitatively the challenges faced when implementing OpenMP on a Software DSM system. We have found that, while results from small test programs have shown promising performance, little information on the behavior of realistic OpenMP applications on Software DSM systems is available. This paper is meant to fill this void. We will pinpoint areas needing improvement and discuss opportunities for addressing the challenges. In particular, we will describe program transformation techniques that may reduce overheads incurred by a naive translation of OpenMP programs onto a Software DSM system.

In our measurements we use a commodity cluster architecture consisting of 16 PentiumII/Linux processors connected via standard Ethernet networks. We expect the scaling behavior of this architecture to be representative of that of common cluster systems with modest network connectivity. We have measured basic OpenMP low-level performance attributes via the kernel benchmarks introduced in [7]. These benchmarks are meant to characterize the performance behavior of OpenMP constructs, such as the time taken to fork and join a parallel region or to execute a barrier. We have also measured a small, highly parallel program, demonstrating upper bounds on the scalability of a Software DSM application. In order to understand the performance characteristics of realistic OpenMP applications, we have measured three of the SPEC OMP2001 applications. We have chosen the applications *wupwise, swim* and *equake*, which are among the most scalable of SPEC's OpenMP benchmarks. *Wupwise* and *swim* are Fortran codes whereas *equake* is written in C. In order to execute these OpenMP programs on a Software DSM system we have hand-translated them into the API provided by the Treadmarks system [8]. We will describe some of the key transformation steps.

The rest of the paper is organized as follows. In Section 2 we will describe the transformations applied to the OpenMP benchmarks in order to execute them on the Treadmarks Software DSM system. In Section 3 we present and discuss our measurements. In Section 4 we describe techniques for improving the performance of OpenMP/Software DSM programs. Section 5 concludes the paper.

2 Translating OpenMP Applications into Software DSM Programs

Common translation methods for loop-parallel languages employ a microtasking scheme. Here the *master processor* begins program execution as a sequential process and pre-forks *helper processes* on the participating processors, which *sleep* until needed. On encountering a parallel construct, the master wakes the helpers and sets up the requisite execution environment. Such microtasking schemes are typically used in shared-memory environments with low communication latencies and fully shared address-spaces.

Software distributed shared-memory systems usually exhibit significantly higher communication latencies and they do not support fully-shared address spaces. In the Treadmarks Software DSM system used in our work, shared-memory sections can be allocated on-request. However, all process-local address spaces are private – not visible to other processes. Typically, there are also constraints on the amount of shared-memory which can be allocated. This is an issue for applications with large data-sets. These properties question the benefit of a microtasking scheme because (1)the helper wakeup call performed by the master process would take significantly longer and (2) the data environment communicated to the helpers could only include addresses from the explicitly allocated shared address space. Therefore, in our work we have chosen an SPMD scheme, as is more common in applications for distributed systems. In an SPMD scheme all processes begin execution of the program in parallel. Sections that need to be executed only by one processor must be marked explicitly and executed conditionally on the process identification.

We translate OpenMP workshare constructs in the usual method, by modifying lower and upper bound of the loops according to the iteration space assigned for each participating process. Currently we support static scheduling only. All parallel constructs are placed between a pair of barrier synchronizations. The barriers perform the dual functions of synchronization and maintaining coherence of shared data. Our SPMD approach is different from that adopted in [1] and it resolves, to an extent, the issues described earlier. We do not need explicit wakeup calls. We may also substitute communication of certain data items by redundantly computing them and thus reduce the amount of memory that must be allocated as shared.

The translation of serial program sections now becomes non-trivial. The need to maintain correct control flow precludes the possibility of executing the serial section by only the master process. Thus, following the SPMD method, parts of the serial section are redundantly executed by all participating processes. However, correctness constraints dictate that shared-memory updates as well as certain operations like I/O in the serial section be done only by the master process. Each such portions is marked to be executed by the master only and is followed by a barrier to ensure shared-memory coherence.

3 Performance Measurements

In this section we describe and discuss some measurements carried out on the performance of OpenMP kernels, microbenchmarks and real-application benchmarks on our cluster. The programs were hand-translated using the transformations described in Section 2, without any further optimizations.

3.1 Speedup Bounds: Performance of an OpenMP Kernel

In order to test the basic capability of scaling parallel code, we measured the performance of a simple OpenMP kernel, which is a small, highly parallel program that calculates the value of π using the integral approximation $\int_0^1 \frac{4.0}{(1.0+x^2)}\,dx$. The OpenMP constructs used within the program include a single OMP DO with a REDUCTION clause. The execution times and speedups obtained(shown in Figure 1) provides an upper bound on the performance we can expect for our system. This kernel makes use of only 3 shared scalar variables.Thus the coherence actions at the barrier terminating the parallel region are minor. This point will assume significance when we compare the performance of this kernel to real applications.

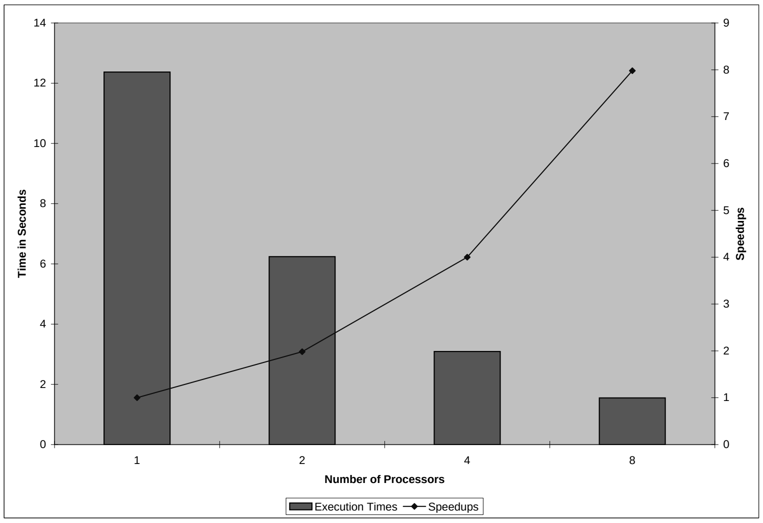

Fig. 1. Execution Time and Speedup of the *PI* program on 1,2,4 and 8 processors

3.2 Performance of the OpenMP Synchronization Microbenchmark

In order to understand the overheads incurred by the different OpenMP constructs on our system, we have used the kernel benchmarks introduced in [7]. In the present work we are primarily interested in the synchronization overheads. The performance of our system on the Synchronization Microbenchmark is shown in Figure 2.

The trends displayed in Figure 2 look similar to those for the NUMA machines (such as the SGI Origin 2000) enumerated in [7]. This is consistent with the fact that a NUMA machine is conceptually similar to a Software DSM system. However, for a Software DSM system, the overheads are now in the order

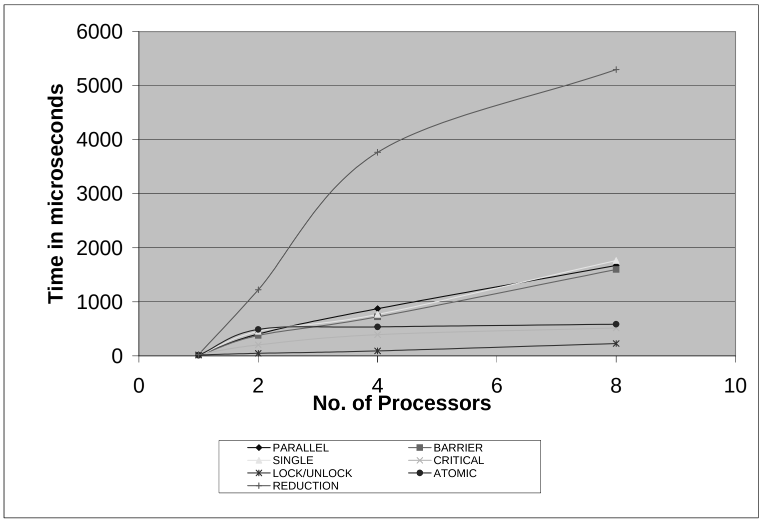

Fig. 2. OpenMP Synchronization Overheads as measured by the OpenMP Synchronization Microbenchmark. The overheads have been measured on a system of 1,2,4 and 8 processors.

of milliseconds as compared to microseconds overhead in NUMA SMPs. The dominant factor is the *barrier* overhead, since the barrier also maintains coherence of the shared-memory. These coherence overheads grow with the increase in shared memory activity within parallel constructs in a Software DSM. This fact is not captured here since the Synchronization Microbenchmark, like the *PI* kernel benchmark, uses a very small amount of shared memory. To quantitatively understand the behavior of OpenMP programs that utilize shared data substantially, we look at three of the most scalable applications in the SPEC OMP2001 benchmark suite.

3.3 Performance Evaluation of Real Application Benchmarks

The SPEC OMP2001 suite of benchmarks [9] consists of realistic OpenMP C and Fortran applications. We selected three of the more scalable benchmarks, namely $SWIM$ and $WUPWISE$ from the set of Fortran applications and $EQUAKE$ from the set of C applications. The trends in our performance measurements was found to be consistent for all three applications. So, for brevity, we shall primarily limit our discussion of exact measurements to the $EQUAKE$ benchmark.

Execution Time Figure 3 shows the execution times for the $EQUAKE$ benchmark run using the *train* dataset. The times shown here include the startup time. For each processor, the total elapsed time has been expressed as a sum of the user and the system times.

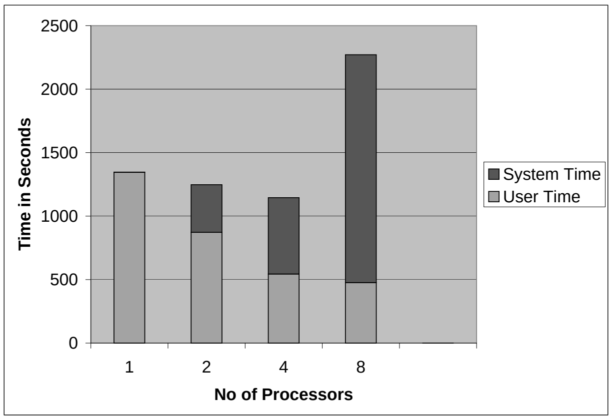

Fig. 3. Execution Times for the *equake* SPEC OpenMP Benchmark running on 1,2,4 and 8 processors, shown in terms of the measured user times and system times.

The figure shows a speedup in terms of user-times from one to eight processors. However, considering total elapsed time, the overall speedup is much less. For 8 processors, the performance degrades so that no speedup is achieved. This fact is consistent with the growing system time component as we go from serial to 8 processor execution. We verified that this system time component is not caused by system scheduling or paging activity. Instead, we attribute the system time to shared-memory coherence activities.

Detailed Measurements of Program Sections The *equake* code has two small serial sections and several parallel regions, which cover more than 99% of the serial execution time. We first look at two parallel loops $main\_2$ and $main\_3$ which show the desirable speedup behavior. The times spent within the parallel loop and the time spent at the barrier are shown separately. Loop $main\_3$ is one of the more important parallel loops where around 30% of the program time is spent. Figure 4 shows the behavior of these loops.

Loop $main\_2$ shows a speedup of about two and loop $main\_3$ shows almost linear speedup on 8 processors in terms of elapsed time. In both cases the loop time itself speeds up almost linearly, but the barrier time increases considerably. In fact, $main\_2$ has a considerable increase in barrier overheads from the serial to 8 processor execution. In $main\_3$, the barrier overhead is within a more acceptable range. The reason for this is that the loop body of $main\_2$ is very small. It consists of a loop of 3 iterations containing a single assignment statement. Compared to that, $main\_3$ has a larger loop body which considerably amortizes the cost of the barrier.

We next look at three loops, $main\_1$ $meminit\_1$ and $meminit\_2$, which incur large barrier overheads. Figure 5 shows the behavior of these loops.

In all these three loops, loop times speed up linearly, but the barrier times increase to 15 to 30 times the loop times on 8 processors. So, for these loops,

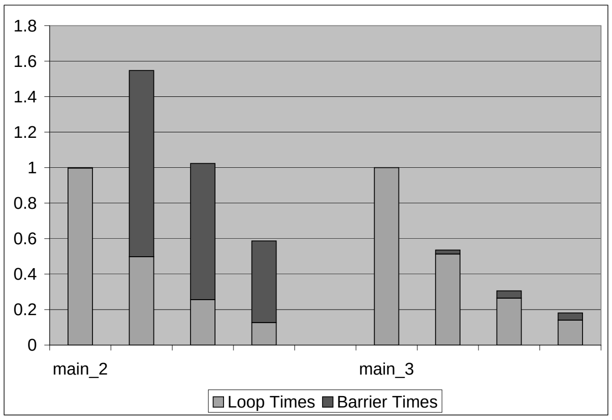

Fig. 4. Normalized Execution Times for the parallel loops, *main_2* and *main_3*, on 1,2,4 and 8 processors. The total execution time for each is expressed as a sum of the time spent within the loop and the time spent on the barrier at its end. Timings for each loop are normalized with respect to the serial version.

there is a substantial slowdown as we go from serial to 8-processor execution. These loops mainly contain shared memory writes. Especially *meminit_1* and *meminit_2* contain only shared memory initialization. So the barrier time becomes unacceptably more expensive than the loops themselves. As discussed previously, the barrier cost increases with the number of shared memory writes in the parallel region.

We next look at the performance of the serial sections (*serial_1* and *serial_2*) and the most time consuming parallel region ($smvp_0$) within the program. Figure 6 shows the behavior of these regions.

As previously discussed in Section 2, barriers have to be placed within a serial region so that shared memory writes by the master processor become visible to the other processors. We see that this results in serial section *serial_1* experiencing a slowdown as we go from the serial to 8-processor execution. Section *serial_2* in unaffected since it does not have barriers in-between.

The parallel region *smvp_0* also suffers because of the same reason - it contains several parallel loops and after each of these, a barrier has to be placed. The parallel region *svmp_0* takes up more than 60% of the total execution time, hence its performance has a major impact on the overall application.

To summarize our measurements, we note that a naive transformation of realistic OpenMP applications for Software DSM execution does not give us the desired performance. A large part of this performance degradation is owing to the fact that shared-memory activity and synchronization is more expensive in a Software DSM scenario and this cost is several orders of magnitude higher than in SMP systems. Barrier costs now increase with the number of shared memory writes in parallel regions. Realistic applications use a large shared-memory space,

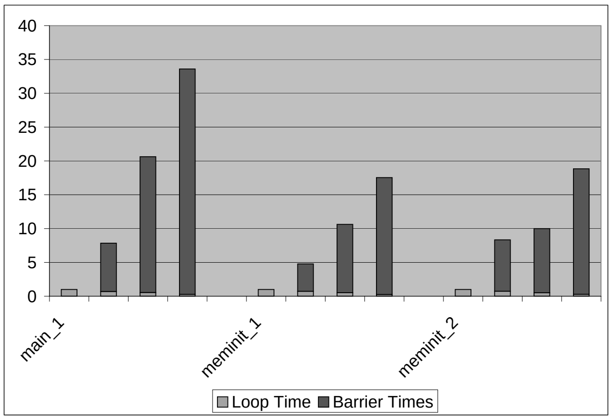

Fig. 5. Normalized Execution Times for the parallel loops *main_1 meminit_1* and *meminit_2* on 1,2,4 and 8 processors. The total execution time for each is expressed as a sum of the time spent within the loop and the time spent on the barrier at its end.

and thus have high data coherence overheads, which explains the seemingly contradicting performance results of kernels and real-applications.

Our measurement motivate the need for optimizations when we try to run real OpenMP applications in a Software DSM scenario. The results discussed above indicate that these optimizations should be primarily aimed at optimizing shared-memory synchronization in programs, irrespective of the underlying Software DSM specifics. In our subsequent discussion, we present some of the possible optimizations.

4 Ongoing Work: Improving OpenMP Performance on Software DSM Systems

In this section we describe transformations that can optimize OpenMP programs on a Software DSM system. Many of these techniques have been discussed in other contexts. We expect their combined implementation in an OpenMP/Software DSM compiler to have a significant impact. The realization of such a compiler is one objective of our ongoing project.

Data Prefetch and Forwarding: Prefetch is a fundamental technique for overcoming memory access latencies. Closely-related to prefetch is *data forwarding*, which is producer-initiated. Forwarding has the advantage that it is a one-way communication (producer forwards to all consumers) whereas prefetching is a two-way communication (consumers request data and producers respond). An important issue in prefetch/forwarding is to determine the optimal prefetch point. Prefetch techniques have been studied previously [10], albeit not in the context of OpenMP applications for Software DSM systems. We expect prefetch/forwarding

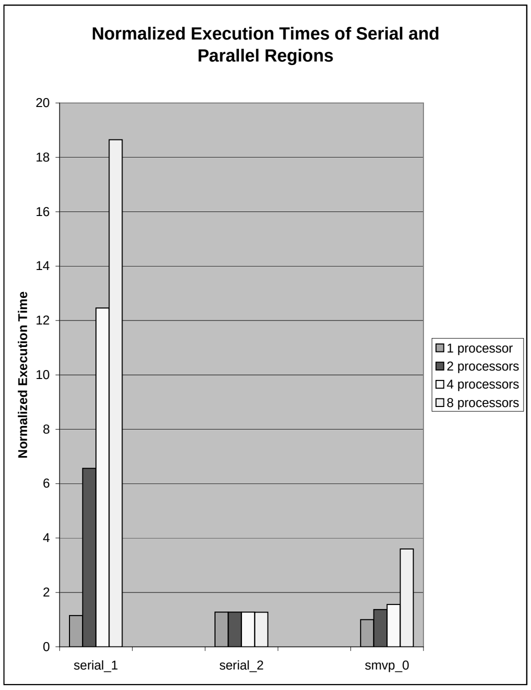

Fig. 6. Normalized Execution Times for the serial sections *serial_1*, *serial_2* and the parallel region *smvp_0* . For each region the times are normalized with respect to the serial version.

to significantly lower the cost of OpenMP END PARALLEL region constructs, as it reduces the need for coherence actions at that point.

Barrier elimination: Two types of barrier eliminations will become important. It is well known that the usual barrier that separates two consecutive parallel loops can be eliminated if permitted by data dependences [11]. Similarly, within parallel regions containing consecutive parallel loops, it may be possible to eliminate the barrier separating the individual loops. Barriers in the serial section, as described in Section 2, can be eliminated in some situations, if the shared data written are not read subsequently within the same serial section.

Data privatization: Private data in a Software DSM system are not subject to costly coherence actions. An extreme of a Software DSM program can be viewed as a program that has only private, and no shared, data. The necessary data exchanges between processors are performed by copy operations via shared memory or possibly via explicit messages. Such a program is equivalent to a message-passing program. Many points are possible in-between this extreme and a program that has all OpenMP shared data placed in shared DSM address space. For example, shared data with read-only accesses in certain program sections can be made "private with copy-in" during these sections. Similarly, shared data that are exclusively accessed by the same processor can be privatized during such a program phase. For large phases, the benefit of this scheme is obvious. In programs with small read-only or exclusive phases, the copy-in and copy-out operations at phase boundaries become significant, moving closer the the "message-passing" extreme mentioned above.

Figure 7 demonstrates a simple form of this optimization, applied to the *equake* benchmark. The program reads data in a serial region from a file. In the original code, since all the processors use the data, a shared attribute is given. However, we have modified the program so that the input data, which are read-only after the initialization, become private to all processes. Figure 7 shows that even this simple optimization substantially reduces the execution times. With this new scheme, the resulting speedup on four processors is close to two.

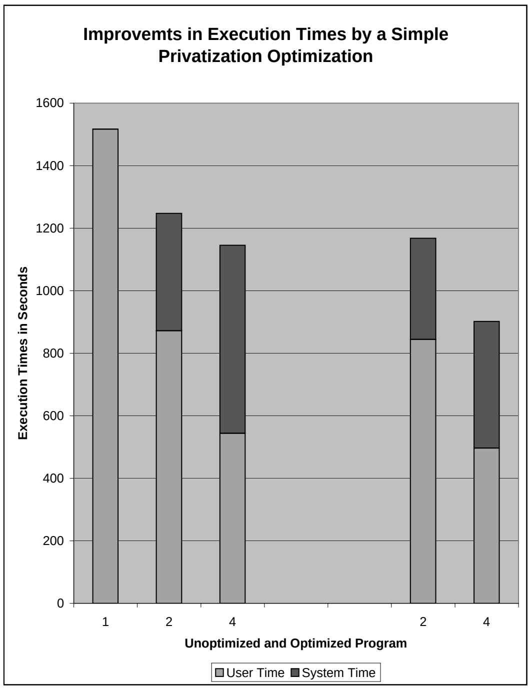

Fig. 7. Performance Improvements with a simple data privatization optimization

Page placement: Software DSM systems may place memory pages on fixed home processors or the pages may migrate between processors. Fixed page placement leads to high overhead if the chosen home is not the processor with the most frequent accesses to this page. Migrating pages can incur high overhead if the pages end up changing their home frequently. In both cases the compiler can help direct the page placing mechanism. It can estimate the number of accesses made to a page by all processors and choose the best home.

Automatic data distribution: Data distribution mechanisms have been well researched in the context of distributed memory multiprocessors and languages such as HPF. Many of these techniques are directly applicable to a Software DSM system. Automatic data distribution is easy in regular data structures and program patterns. Hence, a possible strategy is to apply data distribution with explicit messaging in regular program sections and rely on Software DSM mechanism for other data and program patterns. This idea has been pursued in [12], and we will combine our approaches in a collaborative effort. Planned enhancements include other data and loop transformation techniques for locality enhancement.

Adaptive optimization: A big impediment for all compiler optimizations is the fact that input data is not known at compile-time. Consequently, compilers must make *conservative assumptions* leading to suboptimal program performance. The potential performance degradation is especially high if the compiler's decision making chooses between transformation variants whose performance differs substantially. This is the case in Software DSM systems, which typically possess several tunable parameters. We are building on a prototype of a dynamically adaptive compilation system [13–15], called ADAPT, which can dynamically compile program variants, substitute them at runtime, and evaluate them in the executing application.

In ongoing work we are creating a compilation system that integrates the presented techniques. As we have described, several of these techniques have been proposed previously. However, no quantitative data of their value on large, realistic OpenMP applications on Software DSM systems is available. Evaluating these optimizations in the context of the SPEC OMP applications is an important objective of our current project.

Our work complements efforts to extend OpenMP with latency management constructs. While our primary emphasis is on the development and evaluation of compiler techniques, we will consider small extensions that may become critical in directing compiler optimizations.

5 Conclusions

In this paper we have studied the feasibility of executing OpenMP applications on a commodity cluster architecture via a distributed shared memory system. We have found that, although small OpenMP programs can exhibit near-ideal speedups, a naive translation of realistic OpenMP programs onto the API provided by the Software DSM system is not sufficient. We have found that coherence actions performed at synchronization points are major sources of overheads. We found that the efficient execution of realistic OpenMP applications on distributed processor architectures via a Software DSM system requires substantial program optimizations, one of which we demonstrated through a simple experiment. We are in the process of realizing such optimizations in a new optimizing compiler.

References

1. Y.C. Hu, H. Lu, A.L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, December 2000.
2. J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA Machines. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC2000)*, November 2000.
3. V. Schuster and D. Miles. Distributed OpenMP, Extensions to OpenMP for SMP Clusters. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000)*, July 2000.
4. T.S. Abdelrahman and T.N. Wong. Compiler support for data distribution on NUMA multiprocessors. *Journal of Supercomputing*, 12(4):349–371, October 1998.
5. Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. In *Workshop on OpenMP Applications and Tools (WOMPAT2001)*, July 2001.
6. Lorna Smith and Mark Bull. Development of Mixed Mode MPI / OpenMP Applications. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000)*, July 2000.
7. J.M.Bull. Measuring Synchronization And Scheduling Overheads in OpenMP. In *Proc. of the European Workshop on OpenMP (EWOMP99)*, October 1999.
8. 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, February 1996.
9. V.Aslot, M.Domeika, R.Eigenmann, G.Gaertner, W.B. Jones, and B.Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001), Lecture Notes in Computer Science, 2104*, pages 1–10, July 2001.
10. E.H.Gornish, E.D.Granston, and A.V.Veidenbaum. Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies . *Proceedings of ICS'90, Amsterdam, The Netherlands*, 1:342–353, June 1990.
11. C.W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proc. of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, July 1995.
12. J.Zhu and J.Hoeflinger. Compiling for a Hybrid Programming Model Using the LMAD Representation. In *Proc. of the 14th annual workshop on Languages and Compilers for Parallel Computing (LCPC2001)*, August 2001.
13. M.J.Voss and R.Eigenmann. High-level adaptive program optimization with adapt. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 93 – 102. ACM Press, June 2001.
14. M.J.Voss and R.Eigenmann. A framework for remote dynamic program optimization. In *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, January 1999.
15. M.J.Voss and R.Eigenmann. Dynamically adaptive parallel programs. In *International Symposium on High Performance Computing*, pages 109–120, Kyoto, Japan, May 1999.