

Supporting Realistic OpenMP Applications on a Commodity Cluster of Workstations

Seung Jai Min, Ayon Basumallik, 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 present techniques for translating and optimizing realistic OpenMP applications on distributed systems. The goal of our project is to quantify the degree to which OpenMP can be extended to distributed systems and to develop supporting compiler techniques. Our present compiler techniques translate OpenMP programs into a form suitable for execution on a Software DSM system. We have implemented a compiler that performs this basic translation, and we have proposed optimization techniques that improve the baseline performance of OpenMP applications on distributed computer systems. Our results show that, while kernel benchmarks can show high efficiency for OpenMP programs on distributed systems, full applications need careful consideration of shared data access patterns. A naive translation (similar to the basic translation done by OpenMP compilers for SMPs) leads to acceptable performance in very few applications. We propose optimizations such as *computation repartitioning*, *page-aware optimizations*, and *access privatization* that result in average 70% performance improvement on the SPEC OMPM2001 benchmark applications.

Keywords : OpenMP Applications, Software Distributed Shared Memory, benchmarks, performance characteristics, optimizations

1 Introduction

OpenMP [1] has established itself as an important method and language extension for programming shared-memory parallel computers. While OpenMP has clear advantages on shared-memory platforms, message passing is today still the most widely-used programming paradigm for distributed-memory computers. In this paper, we explore the suitability of OpenMP for distributed systems as well.

* This material is based upon work supported in part by the National Science Foundation under Grant No. 9703180, 9975275, 9986020, and 9974976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Our basic approach is to use a Software DSM (Distributed Shared Memory) system, which provides the view of a shared address space on top of a distributed-memory architecture. To this end, we have implemented a compiler that transforms OpenMP programs into Treadmarks Software DSM programs [2]. This paper makes the following specific contributions.

- We describe our compiler infrastructure that can translate OpenMP applications into Software DSM programs.
- We measure the baseline performance of the application-level SPEC OMPM2001 benchmarks on a distributed memory system, and we analyze the performance behavior.
- We present optimization techniques that enhance the baseline performance of real OpenMP applications on distributed memory system.

Our work is closely related to the following projects. In [3], the authors implemented OpenMP on a network of shared memory multiprocessors and showed their performance using a subset of the SPLASH-2 and NAS benchmark suites, without the help of compiler optimization. In [4], compiler optimizations have been introduced for OpenMP programs on Software DSM. Several recent papers have proposed language extensions. For example, in [5–7], the authors describe data distribution directives similar to the ones designed for High-Performance Fortran (HPF) [8].

From these related efforts, we found that, while results from small kernel programs have shown promising performance, little information on the behavior of realistic OpenMP applications on Software DSM systems is available. In this paper, we show how application-level benchmarks performs on Software DSM and propose optimization techniques to improve the speedups. Also, the optimization techniques that we are presenting in this paper use standard OpenMP as input and do not rely on the user’s data distribution input.

The paper is organized as follows. Section 2 will present basic compiler techniques for translating OpenMP into Software DSM programs. Section 3 will discuss the performance behavior of such programs. Section 4 will present advanced optimizations. In Section 5, we will quantitatively evaluate our proposed techniques, followed by conclusions in Section 6.

2 Translating OpenMP Applications into Software DSM Programs

In the transition from shared-memory to distributed systems, the major challenge is that each participating node now has its own private address space, which is not visible to other nodes. This difference in address spaces affects the OpenMP translation. In case of SMPs, implementing OpenMP shared variables is straightforward, because all variables are accessible by all threads. By contrast, in Software DSM systems, all variables are private by default, and shared data has to be explicitly allocated as such. This creates a challenge for the translation of most OpenMP programs, where variables are shared by default. To address

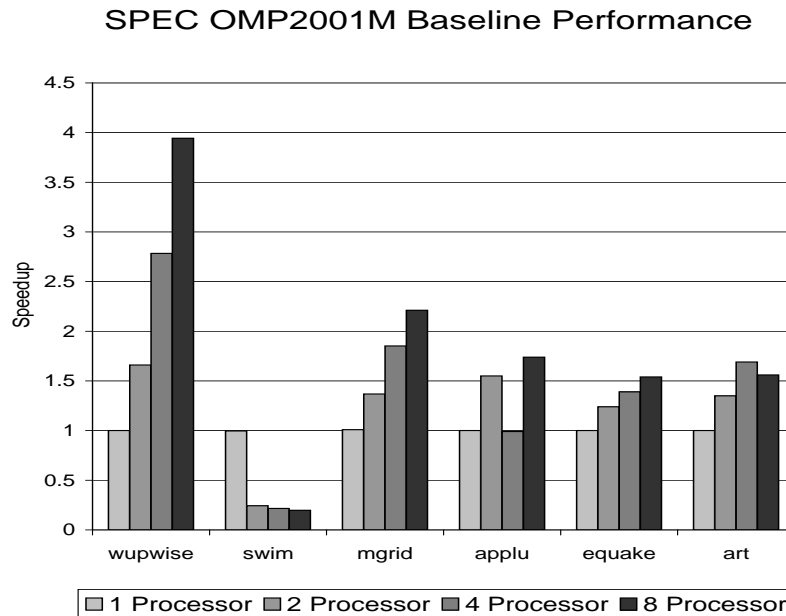


Fig. 1. Baseline performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines.

this challenge, we have developed a compiler infrastructure that performs the following translations. First, our compiler converts the OpenMP application into a micro-tasking form [9]. Second, it converts OpenMP shared data into the form necessary for Software DSM systems. Also, OpenMP shared data may include subroutine-local variables, which reside on the process stack. Stacks on one node are not visible to other Software DSM nodes. We have developed a compiler algorithm that identifies shared variables, using inter-procedural analysis, and changes their declaration to an explicit allocation in shared space.

3 Performance Evaluation of Real Application Benchmarks

In previous work [10], we measured the performance of kernel programs to estimate an upper bound for the performance of OpenMP applications on our system. We also used micro-benchmarks to quantify the performance of specific OpenMP constructs. In this section, we describe and discuss the measurements carried out on the baseline performance of real-application benchmarks on our

cluster. The SPEC OMPM2001 suite of benchmarks [11] consists of realistic OpenMP C and Fortran applications.

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. The baseline performance is illustrated in Figure 1. We translated four SPEC OMPM2001 Fortran programs (WUPWISE, SWIM, MGRID, APPLU) using our compiler and two SPEC OMPM2001 C (ART, EQUAKE) programs by hand. We evaluated the performance on a commodity cluster consisting of PentiumII/Linux nodes, connected via standard Ethernet networks. These benchmark programs are known to exhibit good speedups on shared memory systems [11]. However, their performance on Software DSM systems shows different behavior. For Fortran applications, WUPWISE and MGRID exhibit speedups, whereas the performance of SWIM and APPLU degrades significantly, as the number of processors increases. Evidently, porting well-performing shared-memory programs to Software DSM does not necessarily lead to uniformly good efficiency. A large part of this performance degradation is due to the fact that real-application benchmarks have complex memory access patterns, which causes expensive shared memory activity on Software DSM. In Section 4, we will analyze the causes for performance degradation further and propose optimization techniques for OpenMP programs on Software DSM systems.

4 Advanced Optimizations

The baseline translation of SPEC OMPM2001 programs, described in Section 3, shows that converting shared-memory programs to Software DSM programs requires optimization. Software DSM implementations have shown acceptable performance on kernel programs [2]. However, kernels differ from realistic shared memory applications in two essential ways: (1) in terms of shared data access patterns and (2) in terms of the size of the shared data space.

A realistic application typically consists of several algorithms that access the shared data in different ways. These access patterns may result in increased message traffic in the underlying Software DSM layer, which is expensive from a performance viewpoint. Kernel programs do not exhibit the complex access patterns of full-sized applications and thus do not bring out these additional costs. Secondly, as the size of shared data is increased, we observed that the coherence and update traffic increased significantly. Typical realistic shared memory applications, such as the SPEC OMPM2001 applications, may have data sets that are in the order of gigabyte.

To address the above issues, we have implemented optimizations that fall into three categories.

- Computation repartitioning for locality enhancement
- Page aware optimization techniques
- Shared data space reduction through privatization

<pre> !\$OMP PARALLEL DO DO k=2, nz-1 DO i=ist, iend DO j=jst, jend DO m=1, 5 rsd(m,i,j,k)=... ... ENDDO ENDDO ENDDO ENDDO ... !\$OMP END PARALLEL DO !\$OMP PARALLEL DO DO j=jst, jend DO i=ist, iend DO k=2, nz-1 DO m=1, 5 rtmp(m,k)=rsd(m,i,j,k)-... ... ENDDO ENDDO ENDDO ENDDO ... !\$OMP END PARALLEL DO </pre> <p>(a) original code</p>	<pre> !\$OMP PARALLEL DO k=2, nz-1 DO i=ist, iend !\$OMP DO DO j=jst, jend DO m=1, 5 rsd(m,i,j,k)=... ... ENDDO ENDDO !\$OMP ENDDO ENDDO ENDDO ... !\$OMP END PARALLEL DO !\$OMP PARALLEL DO DO j=jst, jend DO i=ist, iend DO k=2, nz-1 DO m=1, 5 rtmp(m,k)=rsd(m,i,j,k)-... ... ENDDO ENDDO ENDDO ENDDO ... !\$OMP END PARALLEL </pre> <p>(b) after computation repartitioning</p>
--	--

Fig. 2. Computation Repartitioning: subroutine RHS from APPLU

4.1 Computation Repartitioning

Page-based Software DSM systems implement consistency by exchanging information at the page level. Between synchronization points, the participating nodes exchange information about which nodes wrote into each page. A node that writes to a page in shared memory thus becomes the temporary *owner* of that particular page. A page could have multiple temporary *owners* if there are multiple nodes writing to the same page between synchronization points. The way this ownership changes during the program may significantly affect the execution time for the application.

For example, the main loop in APPLU contains seven parallel DO-loops. All these parallel DO-loops access a shared array $rsd(m,i,j,k)$. Five of these seven parallel DO-loops partition array rsd using the outer most index k . One loop does block partitioning, using both i and j indices and the other partitions using the j index. Thus, the main loop of APPLU has four access pattern changes per every loop iteration. Figure 2 illustrates the change in access patterns between two of these loops. This access pattern change will incur a large number of remote node requests in the second parallel loop. To avoid inefficient access patterns, the program needs to be selective about which nodes touch which portions of the data. For example, the code may have a consistent access pattern across loops,

if the inner j -loop is partitioned instead of the outermost k -loop in the first loop nest. Figure 2 (b) shows the resulting code after *computation repartitioning*.

This optimization requires the compiler’s ability to detect further parallelism in the loop nest. We used the Polaris parallelizing compiler for this purpose [12]. However, not all loop nests allow this optimization because some inner loops cannot be parallelized. We applied various techniques, such as adding redundant computation, to enable *computation repartitioning* throughout the whole program.

4.2 Page Aware Optimizations

<pre> !\$OMP PARALLEL DO DO 200 J=1,N DO 200 I=1,M UNEW(I+1,J) = ... VNEW(I,J+1) = ... PNEW(I,J) = ... 200 CONTINUE !\$OMP END PARALLEL DO DO 210 J=1, N UNEW(1,J) = UNEW(M+1,J) VNEW(M+1,J+1) = VNEW(1,J+1) PNEW(M+1,J) = PNEW(1,J) 210 CONTINUE (a)original code </pre>	<pre> !\$OMP PARALLEL DO DO 200 J=1,N DO 200 I=1,M UNEW(I+1,J) = ... VNEW(I,J+1) = ... PNEW(I,J) = ... 200 CONTINUE !\$OMP END PARALLEL DO !\$OMP PARALLEL DO DO 210 J=1, N UNEW(1,J) = UNEW(M+1,J) VNEW(M+1,J+1) = VNEW(1,J+1) PNEW(M+1,J) = PNEW(1,J) 210 CONTINUE !\$OMP END PARALLEL DO (b)after page aware optimization </pre>
--	--

Fig. 3. Page Aware Optimization: subroutine CALC2 from SWIM

Page-aware optimizations use the knowledge that the Software DSM maintains coherence at the page granularity. We will describe two types of *page-aware optimizations*. First, we transform a shared array by *padding*, so that array partitioning across nodes places the partitions at page boundaries. For example, consider an array $U(m, i, j, k)$ of size $U(5, 61, 61, 60)$. The array type is double precision (size of a double precision number is 8 bytes in our system). If the compiler partitions this array U using the index j in the parallel region, then padding the first and the second dimension of U will produce $U(8, 64, 61, 60)$. After padding, the size of the shared array U increases slightly. However, the boundaries of partitioned array chunks are now aligned with the page boundaries. This optimization reduces false sharing around the boundaries of partitioned shared data chunks.

The second *page-aware optimization* deals with the page shape. In a column-major language such as Fortran, a process that writes a column in a two dimensional array will touch much fewer pages than a process that writes a row. As

an example, let A be a 2-D array of size 1024x1024 and its elements are 4 bytes integers. If the size of the page in Software DSM is 4 KB, then each column of A can be mapped to a page. Thus, there are 1024 pages occupying corresponding 1024 columns in A . If a node writes a column in A , then only one page is affected. On the other hand, writing a single row in A touches all the pages owned by all the participating nodes. This scenario is illustrated in Figure 3. In this figure, there is an OpenMP parallel loop followed by a serial loop. In the parallel loop, each node writes to its partitioned blocks of the shared arrays and thus temporarily owns the pages in its partition. Then the master node copies a single row to another row for each shared array in the serial loop and in effect, touches all the pages for these shared arrays. Subsequently, when these shared arrays are read by the child nodes, each child node has to request updates from the master node. This incurs substantial overhead, which can be avoided if the second loop is executed in parallel. In the original benchmark code, the second loop is a serial loop, even though it can be parallelized. This is because parallelizing a small loop is not always profitable in shared memory programming. Thus, this optimization highlights a difference of optimization strategy between shared and distributed memory environments.

4.3 Privatization Optimization

This optimization is aimed at reducing the size of the shared data space that must be managed by the Software DSM system. Potentially, all variables that are read and written within parallel regions are shared variables and must be explicitly declared as such. However, we have found that a significant number of such variables are "read-only" within the parallel regions. Furthermore, we have observed that, for certain shared arrays, different nodes read and write disjoint parts of the array. We refer to these variables as *single-owner* data. In the context of the OpenMP program, these are shared variables. However, in the context of a Software DSM implementation, instances of both can be privatized with certain precautions. The first benefit of privatization stems from the fact that access to a private variable is typically faster than access to a shared variable, even if a locally cached copy of the shared variable is available. This is because accesses to shared variables need to trigger certain coherency and bookkeeping actions within the Software DSM. The second important benefit of privatization is the effect on eliminating false sharing. The overall coherency overhead is also reduced because coherency has to be now maintained for a smaller shared data size.

5 Results

We applied the described optimizations by hand to several of the SPEC OMPM2001 benchmarks and achieved marked performance improvements. Figure 4 shows the final performance obtained by the baseline translation and subsequent optimizations. We present the speedups for four Fortran codes (WUPWISE, SWIM, MGRID, APPLU) and two C benchmarks (EQUAKE, ART).

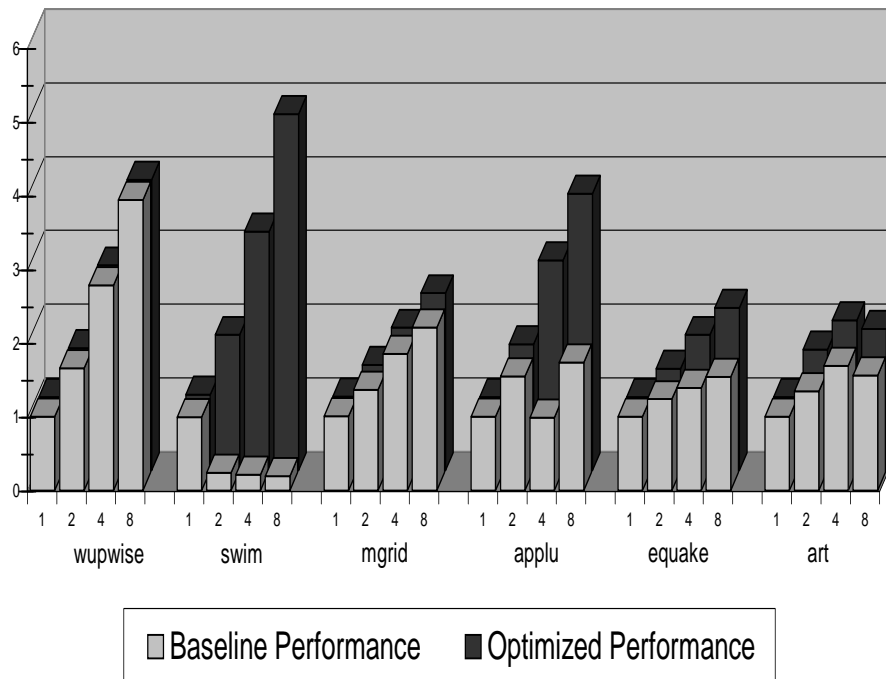


Fig. 4. Performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines.

In one of the Fortran codes, WUPWISE, the baseline translation already had acceptable speedup, and so we did not apply further optimizations. For the three other Fortran codes, SWIM, MGRID, and APPLU, we obtained significant performance improvements with *computation repartitioning* and *page-aware optimizations*. In SWIM, application of the *page-aware optimization* shown in figure 3 improved the performance dramatically. We slightly enhanced the performance of MGRID by applying *computation repartitioning*. APPLU, which shows one of the most complex access patterns among the Fortran applications, has been optimized using both *computation repartitioning* and *page-aware optimizations*. The baseline of APPLU performs better with two than with four processors. In this application, the shared array $rsd(m, i, j, k)$ is block partitioned using both i and j indices so that rsd is partitioned according to the index j on two proces-

sors, and is further partitioned using the index i on four processors, and again using the index j on eight processors and so on. Partitioning using the index i results in multiple nodes writing to a single page and thus causes false sharing. Therefore, whenever *rsd* is partitioned according to the inner index i , it suffers a performance drop owing to the increased false sharing. Since the optimized version for APPLU always partitions *rsd* using the index j , it not only avoids this inconsistent speedup, but also shows much better overall performance.

For the C applications, ART showed improved performance after privatizing several arrays that were not declared as private in the original code. In EQUAKE, we derived benefit from making certain parallel loops dynamically scheduled, though the original OpenMP directives specified static scheduling.

The average baseline speedup of six applications is 1.87 and the average optimized performance is 3.17 on 8 processors. Thus, our proposed optimizations, *computation repartitioning*, *page-aware optimizations*, and *access privatization* result in average 70% performance improvement on our SPEC OMPM2001 benchmarks.

6 Conclusions

In this paper, we have described our experiences with the automatic translation and further hand-optimization of realistic OpenMP applications on a commodity cluster of workstations. We have demonstrated that, for these applications, the OpenMP programming paradigm may be extended to distributed systems. We have discussed issues arising in the automatic translation of OpenMP applications for Software DSM. We have then presented several program optimizations for page-based Software DSM systems. In our performance studies, we have found that a baseline compiler translation, similar to the one used for OpenMP on SMP machines, yields speedups for some of the codes but unacceptable performance for others. After applying the proposed optimizations - *computation repartitioning*, *page-aware optimizations*, and *access privatization* - we observed significant improvements in performance. In the next phase of our project, we intend to use explicit message-passing in conjunction with Software DSM and investigate the effects of our optimizations in this hybrid approach.

References

1. OpenMP Forum, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," Tech. Rep., Oct. 1997.
2. S. Dwarkadas P. Keleher H. Lu R. Rajamony W. Yu C. Amza, A.L. Cox and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18-28, February 1996.
3. H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on network of workstations," in *Proc. of Supercomputing'98*, 1998.
4. Mitsuhsa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi, "OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP," in

Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001), July 2001.

5. R. Crowell Z. Cvetanovic J. Harris C. Nelson J. Bircsak, P. Craig and C. Offner, "Extending OpenMP for NUMA Machines," in *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, November 2000.
6. V. Schuster and D. Miles, "Distributed OpenMP, Extensions to OpenMP for SMP Clusters," in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, July 2000.
7. T.S. Abdelrahman and T.N. Wong, "Compiler support for data distribution on NUMA multiprocessors," *Journal of Supercomputing*, vol. 12, no. 4, pp. 349–371, oct 1998.
8. High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Tech. Rep. CRPC-TR92225, Houston, Tex., 1993.
9. M. Booth and K. Misegades, "Microtasking: A New Way to Harness Multiprocessors," *Cray Channels*, pp. 24–27, 1986.
10. Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," in *Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*. May 2002, Lecture Notes in Computer Science, 2327, Springer Verlag.
11. Rudolf Eigenmann Greg Gaertner Wesley B. Jones Vishal Aslot, Max Domeika and Bodo Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *Proc. of WOMPAT 2001, Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science, 2104*, July 2001, pp. 1–10.
12. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu, "Parallel programming with Polaris," *IEEE Computer*, pp. 78–82, December 1996.