Programming Distributed Memory Sytems Using OpenMP *

Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann

Purdue University
School of Electrical and Computer Engineering
West Lafayette, IN 47907-1285 USA
{basumall,smin,eigenman}@purdue.edu

Abstract

OpenMP has emerged as an important model and language extension for shared-memory parallel programming. On shared-memory platforms, OpenMP offers an intuitive, incremental approach to parallel programming. In this paper, we present techniques that extend the ease of shared-memory parallel programming in OpenMP to distributed-memory platforms as well.

First, we describe a combined compile-time/runtime system that uses an underlying Software Distributed Shared Memory System and exploits repetitive data access behavior in both regular and irregular program sections. We present a compiler algorithm to detect such repetitive data references and an API to an underlying software distributed shared memory system to orchestrate the learning and pro-active reuse of communication patterns.

Second, we introduce a direct translation of standard OpenMP into MPI message-passing programs for execution on distributed memory systems. We present key concepts and describe techniques to analyze and efficiently handle both regular and irregular accesses to shared data. Finally, we evaluate the performance achieved by our approaches on representative OpenMP applications.

1 Introduction

OpenMP [24] has established itself as an important method and language extension for programming shared-memory parallel computers. On these platforms, OpenMP offers an easier programming model than the currently widely-used message passing paradigm. While OpenMP has clear advantages on shared-memory platforms, message passing is today still the most widely-used programming paradigm for distributed-memory computers, such as clusters and highly-parallel systems. In this paper, we investigate the suitability of OpenMP for distributed systems. There are two approaches to achieve this goal, one is to use a Software Distributed Shared Memory (DSM) system, which provides a shared address space abstraction on top of a distributed-memory architecture and the other is to automatically translate *standard* OpenMP programs directly to Message-Passing programs.

Software DSM Systems have been shown to perform well on a limited class of applications [9] [19]. Software DSMs typically adapt a page-based coherence mechanism that intercepts accesses to remote data, at runtime, and requests the data from its current owner. This mechanism incurs runtime overhead, which can be large in applications that communicate frequently with small amounts of data. In most cases, the overheads are larger than in the corresponding, hand-tuned MPI program versions, although it has been shown that the opposite *can* be the case in irregular applications, where hand tuning is difficult [19] [29]. The overhead incurred by Software DSM systems appears primarily as memory access latency. To reduce this memory access latency in Software DSM, we describe a *combined* compile-time/runtime approach. This approach is motivated by the fact that scientific applications contain program sections that exhibit repetitive data access patterns. We present a compiler algorithm to detect such patterns and an API to an underlying software DSM system to orchestrate the learning and pro-active reuse of communication patterns. We evaluate the combined compile-time/runtime system on a selection of OpenMP applications, exhibiting both regular and irregular data reference patterns, resulting in average performance improvement of 28.1% on 8 processors.

Second, we explain our compiler techniques for the

*This work was supported, in part, by the National Science Foundation under Grants No. 9974976-EIA, 0103582-EIA, and 0429535-CCF. 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.

translation of OpenMP programs directly to Messgae Passing programs (in MPI). To achieve good performance, our translation scheme includes efficient management of shared data as well as advanced handling of irregular accesses. In our scheme, shared data is allocated on all nodes. However, there are no shadow copies or management structures, as needed for software DSMs. Furthermore we envision that for future work, arrays with fully regular accesses will be distributed between nodes by the compiler. Therefore, we refer to our scheme as partial-replication. We have studied a number of OpenMP programs and identified baseline techniques for such a translation as well as optimizations that can improve performance significantly. We present compiler techniques for translating OpenMP programs to MPI under the *partial replication* model and describe a runtime inspection-based scheme for translating irregular OpenMP applications.

The remainder of this paper is organized as follows. Section 2 presents the combined compile-time/runtime system to enhance the performance of OpenMP applications deployed on distributed-memory systems using Software DSM. Section 3 describes the automatic translation of OpenMP programs to MPI programs. Section 4 compares our approaches with related work. Section 5 concludes the paper.

2 Optimizing OpenMP Programs on Software Distributed Shared Memory System

We describe a *combined* compile-time/runtime approach to latency reduction in OpenMP applications, deployed on Software DSM systems. In previous work [21], we have presented basic compiler techniques for deploying OpenMP applications on Software DSM systems. Here, we present additional optimizations, based on the detection of repetitive access patterns for shared data. Both the compiler and the runtime system share the task of data reference analysis. The compiler identifies which shared memory accesses will exhibit repetitive access patterns and when. The runtime system captures information, such as the address and the destination of remote memory accesses and optimizes their communication.

2.1 Compiler Analysis of Repetitive Data References

Data references must meet two criteria to be classified as repetitive accesses in a given loop L. (1) The reference must be made in a basic block that executes repeatedly in a sequence of iterations of L and (2) the involved variable must either be scalar or an array whose subscript expressions are *sequence invariant* in L. To check the first criterion, the algorithm determines the *path conditions* [11] for the basic block and tests for loop invariance in L of these conditions. The gist of checking the second condition is in testing sequence invariance of all array subscripts. Sequence invariance of an expression means that the expression assumes the same sequence of values in each iteration of L. (Invariance is a simple case of sequence invariance – the sequence consists of one value.) The output of the compiler algorithm is the set of shared variables that incur repetitive data references across the iterations of L. The detailed algorithm description and examples are illustrated in our previous work [22].

2.2 Compiler/Runtime System Interface

The compiler instruments the code to communicate its findings to the runtime system. It does this through an API containing two functions, which tell the runtime system when/where to learn communication patterns and when/where to repeat them, respectively. The overall model of program execution is important at this point. We assume that the program contains an outer, serial loop (e.g., a time-stepping loop), within which there are a sequence of parallel loops, each terminated by a barrier synchronization. This execution model is consistent with that of many OpenMP programs. We refer to this outer loop as the target loop. The compiler identifies the target loop as the outer-most loop containing parallel sections and a non-empty *RepVar*, the set of shared variables with repetitive reference patterns, described in Section 2.1. Next, it partitions the target loop body into *intervals* – program sections delimited by barrier synchronizations. The API includes the following functions, which the compiler inserts at the beginning of each interval.

– Get_CommSched(*CommSched-id*, *StaticVarList*)

– Run_CommSched(*CommSched-id*)

There are two input parameters – *CommSched-id* is the communication schedule identifier; *StaticVarList* is the list of shared variables with repetitive reference patterns, which will benefit from pro-active data movement at the beginning of the interval. Our compiler generates *StaticVarList* by intersecting the set of shared variables accessed within the interval and *RepVar*. Depending on the value of the index variable of the target loop, either one of these API functions is executed in each iteration. When Get_CommSched is invoked, the runtime system learns the communication pattern of the variables listed in *StaticVarList* and creates a communication schedule, which is the set of data (pages in case of page-based Software DSM) that experienced remote memory access misses. On a call to Run_CommSched, the runtime system finds the communication schedule using *CommSched-id* and pro-actively moves the data according to that schedule.

2.3 The Runtime System

We have modified the TreadMarks [2] version 1.0.3.3 to support the pro-active data movement with message aggregation. The augmented runtime system captures the communication pattern and creates the communication schedule during the interval where *Get_CommSched* is called. On a call to *Run_CommSched*, the augmented runtime system applies the communication schedule by pro-actively moving data. When there are multiple messages to the same processor, those messages are aggregated into a single message to reduce the number of messages communicated. Also, the shared data that are pro-actively moved will not incur remote memory misses during the execution, which results in the reduction of DSM coherence overheads.

2.4 Performance Evaluation

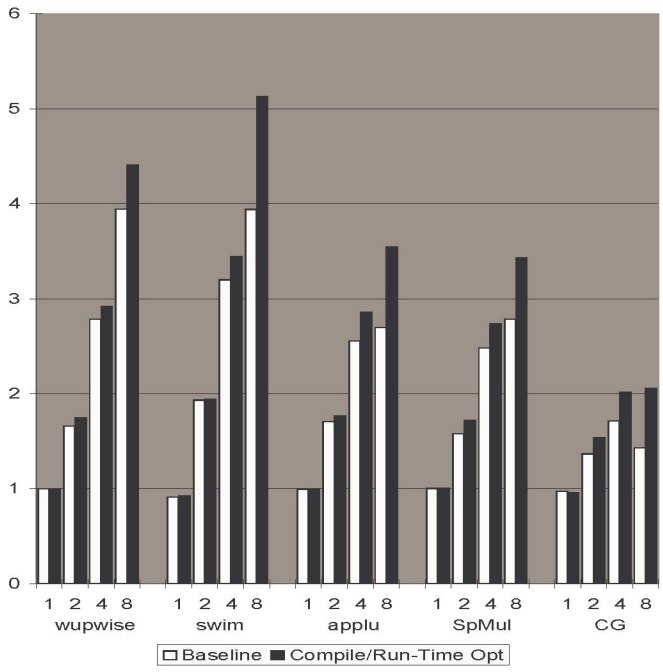

Figure 1. Speedup for TreadMarks and the Combined Compile-time/Run-time System

We evaluated the combined compile-time/runtime system on a selection of SPEC OMP and NAS OpenMP benchmarks, exhibiting both regular and irregular communication patterns. Our commodity cluster consists of Pentium-II/Linux nodes, connected via standard 100Mbps Ethernet networks. We used five Fortran programs: WUPWISE, SWIM, and APPLU from the SPEC OMP benchmarks and CG from the NAS OpenMP benchmarks and SpMul. Among these programs, WUPWISE, SWIM, and APPLU are regular applications and CG and SpMul are mixed regular/irregular applications.

Our PCOMP compiler [23] translates the OpenMP applications into TreadMarks programs. After this transformation, we performed repetitive data reference analysis according to the proposed compiler algorithm and instrumented the programs with the described API functions for pro-active data movement. Overall, our applications show regular communication patterns in most of their execution, even in irregular program sections. For example, CG and SpMul have indirect array accesses. In both applications, the indirection arrays are defined outside each target loops and the compiler analysis is able to determine that the involved array accesses exhibit static communication patterns. Owing to the precise compiler analysis, we can selectively apply pro-active data movement to only shared variables that show regular communication patterns. Figure 1 presents the performance of the baseline TreadMarks and that of the proposed compile-time/runtimes system compared to the sequential execution times. Programs are executed on 1, 2, 4, and 8 processors. On 8 processors, the proposed technique achieves 28.1% performance improvement over the baseline TreadMarks system. The performance enhancement mainly comes from the reduction of the number of messages and the reduction of the page fault overhead. Our compiler analysis makes it possible to obtain these reductions by applying pro-active data movement to the right data at the right time.

3 Translation of OpenMP to MPI

In the previous section, we discussed a combined compile-time/run-time optimization scheme for OpenMP applications deployed through Software DSM systems. However, Software DSM systems suffer from some inherent performance limitations. A comparative study of Message-Passing(using PVM [26]) and TreadMarks applications [20] concluded that message-passing applications have two basic advantages over Software DSM applications. The first advantage is that message-passing applications can piggyback synchronization with sends and receives of data whereas Software DSMs incur separate overheads for synchronization and data transfer. The second advantage is that message-passing applications implicitly perform aggregation for transferring data while Software DSMs are limited by the granularity at which they maintain coherence for shared data(for example, page-based software DSMs perform shared data transfers on a per page basis). Techniques using prefetch [27] and compiler-assisted analysis of future accesses for aggregation [10] have been proposed to mitigate these performance limitations.

In order to avoid the performance limitations imposed

by a Software DSM system, we explore the possibility of translating OpenMP applications directly to message passing applications that use MPI. Essentially, an SDSM layer performs two functions:

- It traps accesses to remote data.
- It provides a mechanism for intercommunicating data between nodes on demand.

For the first function, we now use a combination of compile-time analysis and runtime methods to resolve remote accesses. For the second function, we use MPI libraries to communicate data. The direct use of message passing provides the compiler greater control of when and how processes intercommunicate their data and thus makes it easier to optimize this communication as well as to implement aggregation and prefetching. Additionally, robust and optimized MPI libraries are available for almost all types of distributed-memory platforms.

To achieve good performance, our translation scheme includes efficient management of shared data as well as advanced handling of irregular accesses. In this section, we present a brief overview of the baseline compile-time translation scheme for translating OpenMP applications to MPI and then we present a runtime inspection-based scheme for translating OpenMP applications that have irregular data accesses.

3.1 Baseline OpenMP to MPI Translation Scheme

The objective of the baseline translation scheme is to perform a source-to-source translation of a shared-memory OpenMP program to an MPI program. This is accomplished by two categories of transformations – (1) transformations that interpret the OpenMP directives and (2) transformations that identify and satisfy producer-consumer relationships for shared data.

OpenMP directives fall into three categories -
(1) directives that specify work-sharing (*omp parallel for, omp parallel sections*) - the compiler interprets these to partition work between processes,
(2) directives that specify data properties (*omp shared, omp private* etc.) - these are used for constructing the set of *shared* variables in the program. By default, data is shared as per the OpenMP standard.
(3) synchronization directives (*omp barrier, omp critical, omp flush* etc.) - the compiler incorporates these into the control flow graph.

The target execution model for our translation scheme is SPMD [7] with the following characteristics:

- All participating processes redundantly execute serial regions and parallel regions demarcated by *omp master* and *omp single* directives. Iterations of OpenMP parallel *for* loops are statically partitioned between processes using *block-scheduling*.
- Shared data, is allocated on all processes. There is no concept of an *owner* for any shared data item. There are only producers and consumers of shared data.
- At the end of parallel constructs, each participating process communicates the shared data it has produced that other processes *may* use in the future.

An exception to redundant execution of the serial regions is file I/O. Reading from files is redundantly done by all processes (we assume a file-system visible to all processes). Writing to file is done by only one process (the process with the smallest MPI rank).

After interpreting the OpenMP directives, the compiler needs to inserts MPI calls to communicate shared data from producers to *potential* future consumers. To resolve producer-consumer relationships, the compiler has to perform precise array-dataflow analysis. Several schemes such as Linearized Memory Access Descriptors [25] and Regular Section Descriptors [6] have been proposed to characterize array accesses. Our compiler constructs bounded regular section descriptors [12] to characterize accesses to shared arrays.

The compiler constructs a control flow graph (with each vertex corresponding to a program statement) and records array access summaries with Regular Section Descriptors (RSDs) by annotating the vertices of the control flow graph. The compiler then uses this annotated control flow graph to create a *producer-consumer flow graph* which is used to resolve producer-consumer relationships for shared data. This graph is created by modifying the annotated control-flow graph to conform to the relaxed memory consistency model of OpenMP. OpenMP specifies implicit and explicit memory synchronization points.

The compiler now uses this producer-consumer flow graph to compute message sets for communicating shared data from producers to potential consumers. In previous work [3], we have discussed the algorithm for computing these message sets. The computed message sets are communicated using non-blocking *send/receive* and blocking *wait* calls.

For affine accesses, the compiler can create precise regular section descriptors and thus generate precise message sets for communicating data between producers and consumers. However, in cases where array accesses are not regular, the compiler cannot perform the shared array access analysis precisely at compile-time. We now present run-time mechanisms for translating applications with such irregular accesses.

```
L1 : #pragma omp parallel for
     for(i=0;i<N;i++)
       p[i] = x[i] + alpha*r[i] ;

L2 : #pragma omp parallel for
     for(j=0;j<N;j++) {
       w[j] = 0 ;
       for(k=rowstr[j];k<rowstr[j+1];k++)
S2:        w[j] = w[j] + a[k]*p[col[k]] ;
     }
```

Figure 2. Sparse Matrix-Vector Multiplication Kernel

3.2 Translation of Irregular OpenMP Applications to MPI

Irregular applications pose a challenge, because the compiler cannot accurately analyze the accesses at compile time. Instead, it must conservatively over-estimate data consumption. Consider the following code.

```
L1 : #pragma omp parallel for
     for(i=0;i<10;i++)
       A[i] = ...

L2 : #pragma omp parallel for
     for(j=0;j<20;j++)
       B[j] = A[C[j]] + ...
```

Considering parallel execution on 2 processes (numbered 0 and 1), the compiler summarizes the writes in Loop L1 using an RSD of the form $< p, write, A, 1, 5 * p, 5 * p + 5 >$. For L2, the compilers produces the two RSDs $< p, write, B, 1, 10 * p, 10 * p + 5 >$ and $< p, read, A, 1, undefined, undefined >$. In L2, array A is accessed using the indirection array C and thus, the accesses to A cannot be resolved at compile time. In such cases, our existing compiler [3] attempts to deduce certain characteristics (e.g., monotonicity) for the indirection array A. This information serves to obtain bounds on the region of array A accessed by each process. If no such property can be deduced, our translation scheme will determine that at the end of loop L1, process 0 must send elements A[0] through A[4] to process 1 and process 1 must send elements A[5] through A[9] to process 0, which may result in excess communication. To precisely resolve such irregular accesses, our system makes use of runtime inspection. A key insight is that runtime inspection not only resolves producer consumer relationships precisely for irregular accesses, it also maps accesses to loop iteration. Therefore, to amortize the cost of runtime inspection, we can use this information to reorder iterations for parallel loops to overlap computation and communication.

```
L1 : #pragma omp parallel for
     for(i=0;i<N;i++)
       p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
       for(j=0;j<N;j++) {
         w[j] = 0 ;
       }

L2-2:  #pragma omp parallel for
       for(j=0;j<N;j++) {
         for(k=rowstr[j];k<rowstr[j+1];k++)
S2:          w[j] = w[j] + a[k]*p[col[k]] ;
       }
```

Figure 3. Sparse Matrix-Vector Multiplication Kernel with Loop Distribution of loop L2

```
L3: for(i=0;i<num_iter;i++)
      w[T[i].j] = w[T[i].j] +
                  a[T[i].k]*p[T[i].col] ;
```

Figure 4. Restructuring of Sparse Matrix-Vector Multiplication Loop

Previous work on inspectors in the context of languages such as HPF and Titanium [8, 1, 14] have suggested reordering loop iterations to differentiate local and non-local accesses. Consider a common sparse matrix-vector product shown in Figure 2, taken from the NAS Conjugate Gradient benchmark. The irregular access here occurs in statement S2 inside the nested loop L2. Each outer j iteration in loop L2 now contains multiple irregular accesses to the vector p, which is produced blockwise in loop L1. Therefore, simply reordering the outer j iterations may not suffice to partition accesses into accesses of local and remote data.

To expose the maximum available opportunity for computation-communication overlap, the loop L2 in Figure 2 is restructured to the form shown in Figure 3. Loop L2 is distributed into the loop L2-1 and the perfectly nested loop L2-2. On each process, at run-time, inspection is done for statement S2 and in every execution i of statement S2, the loop indices j and k as well as the corresponding value of $col[k]$ are recorded in an inspection structure T. T here can be considered a mapping function $T : [j, k, col[k]] \rightarrow [i]$. This mapping can then be used to transform loop L2-2 in Figure 3 to loop L3 in Figure 4. Iterations of loop L3 can now be re-ordered to achieve maximum overlap of computation and communication in our sparse matrix-vector multiplication example.

In previous work [4], we have presented the algorithms

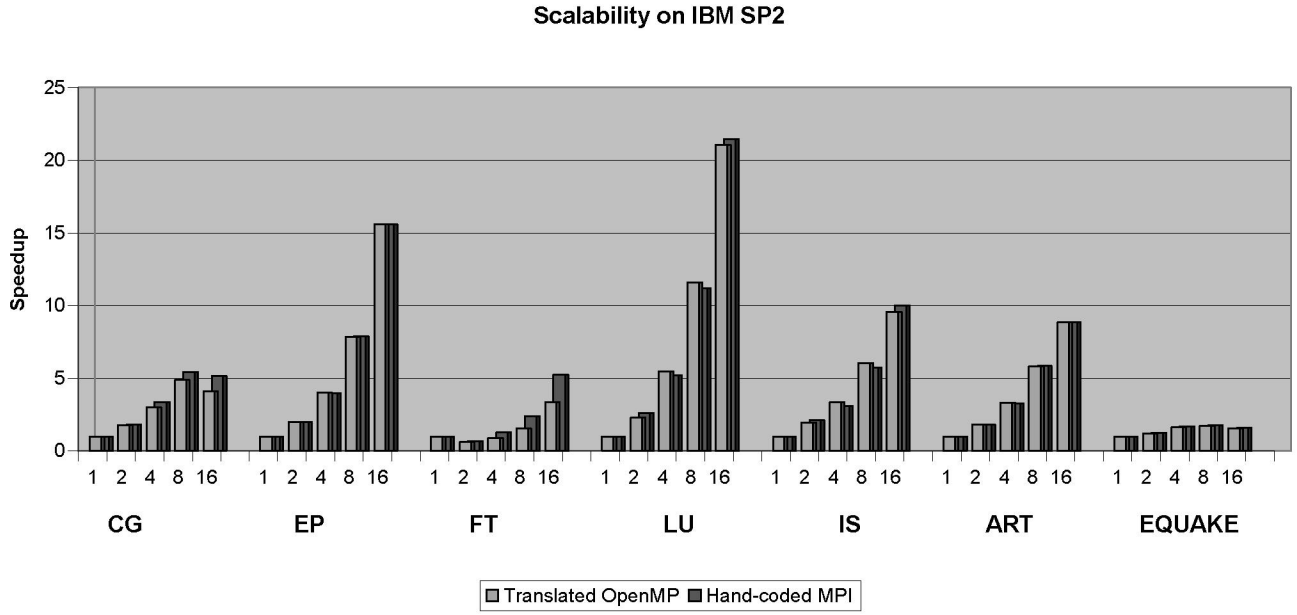

Figure 5. Scalability comparison on the IBM SP2 nodes. The input data set used is CLASS B for all the NAS benchmarks and *train* for the SPECOMPM2001 benchmarks.

to affect these runtime inspection and loop restructuring transformations.

3.3 Performance Evaluation

To evaluate the performance of our translation scheme, we have applied the OpenMP translation steps discussed in this section to seven representative OpenMP applications - five from the NAS Parallel Benchmarks and two SPEC OMPM2001 applications (EQUAKE and ART). Our hardware platform is a set of sixteen IBM SP2 WinterHawk-II nodes connected by a high-performance switch. We expect the scaling behavior of the benchmarks on these systems to be representative of a wide class of high-performance computing platforms. In Figure 5, we compare the scaling behavior of the benchmarks translated to MPI from OpenMP with the scaling behavior of their hand-translated MPI counterparts (for NAS benchmarks, the hand-translated versions are the official MPI versions of these benchmarks. For EQUAKE and ART, we have created the reference MPI verions ourselves with reasonable programming effort). On average, we observed that the translated OpenMP versions have performance to within 15% of their hand translated MPI counterparts. Additionally, the NAS benchmark CG and the SPEC OMPM2001 benchmark EQUAKE contain irregular array accesses in their most time consuming parts. We applied the transformations discussed in Section 3.2 to these applications. The performance of the resulting translation is shown in Figures 6 and 7. We found that with these transformations, the translated OpenMP versions achieved speedups to within 9% of the hand-translated MPI versions.

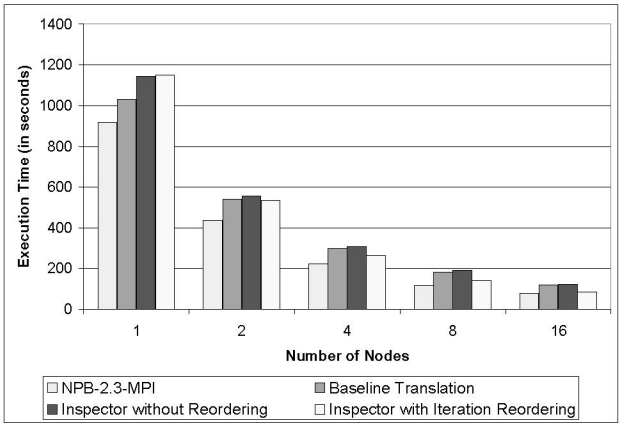

Figure 6. Performance of *CG*.

4 Related Work

Researchers have proposed numerous optimization techniques to reduce remote memory access latency on Software DSM. Many of these optimization techniques aim to perform pro-active data movement by analyzing data access patterns either at compile-time or at runtime. In compile-time methods, a compiler performs reference analysis on source programs and generates information in the form of

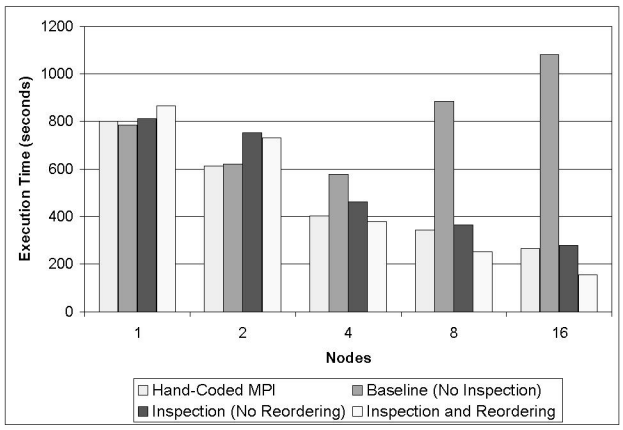

Figure 7. Performance of *Equake*.

directives or prefetch instructions that invoke pro-active data movement at runtime [9]. The challenges for compile-time data reference analysis are the lack of runtime information (such as the program input data) or complex access patterns (such as non-affine expressions). By contrast, runtime-only methods predict remote memory accesses to prefetch data [5] based on recent memory access behavior. These methods learn communication pattern in *all* program sections and thus incur overheads even in those sections that a compiler could recognize as not being beneficial. The idea of combined compile-time/runtime solutions has been expressed by others [28, 16, 15]; however, our paper [22] is the first to present a compiler algorithm and a corresponding application program interface (API), allowing the compiler and runtime system to work in concert.

An important contribution towards a simpler programming model for distributed-memory machines was the development of High Performance Fortran (HPF) [13, 17]. There are important differences between our OpenMP-to-MPI translation approach and that taken by HPF. Even though, like OpenMP, HPF provided directives to specify parallel loops, HPF's focus was on the use of the data distribution directives. Data and computation partitioning was derived from these directives. Most often, data had a single owner and computation was performed on the owning node (a.k.a owner-computes rule). Input operands to the computations were received via messages from their owners. This execution scheme could also add significant overhead to serial sections, as these needed be executed on multiple nodes owning parts of the data. Handling irregular data was difficult and usually employed runtime schemes [8]. In contrast to HPF implementations, our execution model starts from the available parallelism specified through OpenMP directives. Partial replication allows serial regions to be executed intact and input operands of parallel computations are locally available. Communication happens primarily at the end of parallel loops, facilitated by collective communication. Partial replication also obviates the need for data partitioning techniques [18], even though data distribution information is not provided by the user.

5 Conclusions

In this paper, we have discussed two approaches aimed at making OpenMP shared-memory programming available for distributed-memory systems as well. First, we examined a combined compile-time/runtime approach for accelerating the execution of applications with repetitive communication patterns deployed through Software DSM. Our compiler algorithm is essential to accurately and selectively apply pro-active data movement to remote memory accesses showing static communication patterns. We evaluated the proposed compile-time/runtime system using OpenMP applications, consisting of both regular and irregular applications. We achieved performance improvements as significant as 44% and on average 28.1% on 8 processors.

Second, we examined techniques for translating standard OpenMP shared-memory programs directly to a Message Passing form. We discussed the basic OpenMP to MPI translation scheme and the translation of irregular OpenMP applications into MPI codes. We found that the translated OpenMP versions have the performance to within 15% of their hand-translated MPI counterparts.

Our measurements show that the presented techniques, a combined compile-time/runtime technique on Software DSM and direct translation of OpenMP to MPI, significantly improve the performance of OpenMP on distributed memory systems. This fact, combined with the greater ease of programming that OpenMP is generally attributed with, indicates a promising new path toward higher programming productivity on distributed-memory platforms.

References

[1] T. S. Abdelrahman and G. Liu. Overlap of computation and communication on shared-memory networks-of-workstations. *Cluster computing*, pages 35–45, 2001.

[2] 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.

[3] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 189–198, Cambridge, Massachusetts, USA, 2005. ACM Press.

[4] A. Basumallik and R. Eigenmann. Optimizing Irregular Shared-memory Applications for Distributed-memory Systems. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 119–128, New York, NY, USA, 2006. ACM Press.

[5] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software dsms. In *the 12th international conference on Supercomputing*, pages 385–392, 1998.

[6] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *J. Parallel Distrib. Comput.*, 5(5):517–550, 1988.

[7] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[8] R. Das, M. Uysal, J. Saltz, and Y.-S. S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, 1994.

[9] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 186–197, 1996.

[10] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 186–197, 1996.

[11] T. Fahringer and B. Scholz. Symbolic evaluation for parallelizing compilers. In *International Conference on Supercomputing*, pages 261–268, 1997.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[13] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[14] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Technical report, Berkeley, CA, USA, 2001.

[15] P. Keleher. Update protocols and iterative scientific applications. In *Proc. of the first Merged Symp. IPPS/SPDP (IPDPS'98)*, 1998.

[16] P. Keleher and C.-W. Tseng. Enhancing software DSMs for compiler-parallelized applications. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, 1997.

[17] C. Koelbel, D. Loveman, R. Schreiber, G. S. Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[18] U. Kremer. Automatic data layout for distributed memory machines. Technical Report TR96-261, 14, 1996.

[19] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 48–56, 1997.

[20] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, 1997.

[21] S.-J. Min, A. Basumallik, and R. Eigenmann. Optimizing OpenMP programs on Software Distributed Shared Memory Systems. *International Journal of Parallel Programming*, 31(3):225–249, June 2003.

[22] S.-J. Min and R. Eigenmann. Combined compile-time and runtime-driven, pro-active data movement in software dsm systems. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–6, New York, NY, USA, 2004. ACM Press.

[23] S. J. Min, S. W. Kim, M. Voss, S. I. Lee, and R. Eigenmann. Portable compilers for openmp. In *International Workshop on OpenMP Applications and Tools (WOMPAT'01)*, pages 11–19, July 2001.

[24] OpenMP Forum. Openmp: A proposed industry standard api for shared memory programming. Technical report, Oct. 1997.

[25] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[26] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[27] A. K. W. L. Todd C.Mowry, Charles Q. C. Chan. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA'98)*, Feb. 1998.

[28] G. Viswanathan and J. R. Larus. Compiler-directed shared-memory communication for iterative parallel applications. In *Supercomputing*, Nov. 1996.

[29] J. Zhu, J. Hoeflinger, and D. Padua. A synthesis of memory mechanisms for distributed architectures. In *Proceedings of the 15th international conference on Supercomputing*, pages 13–22. ACM Press, 2001.