Towards Automatic Translation of OpenMP to MPI

Abstract. We present compiler techniques for translating OpenMP shared-memory parallel applications into MPI message-passing programs for execution on distributed memory systems. This translation aims to extend the ease of creating parallel applications with OpenMP to a wider variety of platforms, such as commodity cluster systems. We present key concepts and describe techniques to analyze and efficiently handle both regular and irregular accesses to shared data.

We evaluate the performance achieved by our translation scheme on seven representative OpenMP applications, two from SPEC OMPM2001 and five from the NAS Parallel Benchmarks suite, on two different platforms. The average scalability (execution time relative to the serial version) achieved is within 12% of that achieved by corresponding hand-tuned MPI applications. We also compare our programs with versions deployed for a Software Distributed Shared Memory system and find that the direct translation to MPI achieves up to 30% higher scalability. A comparison with High Performance Fortran (HPF) versions of two NAS benchmarks indicates that our translated OpenMP versions achieve 12% to 89% better performance than the HPF versions.

Keywords: Compiler Techniques, OpenMP, MPI, Performance, Commodity Clusters.

1 Introduction

OpenMP [1] 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. Programs written in OpenMP can usually be parallelized stepwise, starting from a sequential program. OpenMP programs often resemble their original, sequential versions, the main difference being the inserted OpenMP directives.

This approach contrasts with using the Message Passing Interface (MPI) [2], which generally requires programs to be translated into parallel form as a whole, often causing them to look drastically different from their original version. Additionally, MPI places the burden of data-to-process mapping and remote access on the programmer, since the only mechanism for data management is the judicious use of explicit messages. Management of logically shared-data, coupled with the task of parallelizing the application as a whole, can make programming in MPI very effort-intensive.

While OpenMP has clear advantages in terms of ease of programming on shared-memory platforms, message-passing is today still the predominant programming paradigm available for distributed-memory computers, including clusters and most of the highly-parallel systems. In this paper, we explore the suitability of OpenMP for distributed systems as well. Our approach is to automatically translate *standard* OpenMP programs directly to MPI programs. To this end, we have investigated compiler techniques for the translation of OpenMP programs to MPI.

To achieve good performance, our translation scheme includes efficient management of shared data as well as advanced handling of irregular accesses. These issues were also considered by related approaches to compile shared-address-space languages onto distributed memory machines. Compiler techniques for High Performance Fortran (HPF) [3] and those for shared-memory programming on clusters using Software Distributed Shared Memory (DSM) Systems are important representatives. A key difference that distinguishes our approach from these others is the concept of *partial replication*. In HPF-like approaches, the focus was on the availability of data distribution directives. Data partitioning, computation partitioning, and message insertion 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). Software DSM systems, such as TreadMarks [4], take a contrasting approach. They not only replicate all shared program data but also allocate management data structures (such as shadow copies of shared data) on all processors. In our scheme, shared data is allocated on all nodes. However, no shadow copies or management structures need to be allocated. 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. Data replication comes at the cost of limiting the data scalability of programs – that is, programs will not be able to handle n-times larger data sets on n processors. While this is a limitation of our scheme, by *partially replicating* data, our method simplifies data management and communication to the point where our techniques outperform HPF programs substantially. At the same time, our system is able to run the large data sets of our benchmarks, which was not possible on software DSMs due to their extensive 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. This paper makes the following contributions:

- We present compiler techniques for translating OpenMP programs to MPI under the *partial replication* model, including
  - the discussion of OpenMP semantics under the new model.

- an algorithm for computing message sets,
- techniques for statically handling irregular accesses, and
- optimizations based on collective communication.

– We evaluate seven representative OpenMP programs from the NAS and SPEC OMP benchmarks on two platforms. We compare the performance of our new translation scheme with corresponding hand-tuned MPI program versions and with programs optimized for SDSM platforms. We also compare the performance of available HPF program versions and add qualitative comparisons with related paradigms (such as UPC [5]).

The rest of the paper is organized as follows. Section 2 presents the OpenMP to MPI baseline translation and discusses OpenMP semantics and message set computation under the partial replication model. It also presents basic optimizations. Section 3 presents techniques for handling irregular accesses at compile-time. Section 4 presents a set of compiler optimizations that significantly improve the performance of the translated applications. Section 5 presents an evaluation of the performance achieved by a set of representative OpenMP applications translated using these techniques on two different platforms. Section 6 places this paper in the context of related work. Section 7 concludes the paper.

2 OpenMP to MPI Baseline Translation

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. We have implemented these compiler transformations using the Cetus compiler infrastructure [6].

2.1 Interpretation of OpenMP Semantics Under Partial Replication

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 execution model under partial replication 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 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.

The method of producers forwarding data to all *potential* consumers ensures that processes always make local accesses when they read shared data. This method may result in redundant communication where accesses are not completely analyzable. Section 5 will show that our techniques perform well for several representative applications. Furthermore, we present a technique for refining irregular accesses in Section 3.

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).

Our compiler converts OpenMP *section* constructs to OpenMP loops using a *switch-case* construct so that each iteration executes a different OpenMP section. Henceforth, the term "work-sharing construct" will only refer to OpenMP loops. OpenMP has four scheduling strategies that may be specified for parallel loops - static, dynamic, guided and runtime. Currently, we support only static scheduling. Additionally, at this step, if the OpenMP loop contains a *reduction* clause, then the compiler replaces operations on the reduction variable with operations on a temporary variable and inserts an *MPI_Allreduce* function call to reduce the value of the temporaries into the reduction variable.

In the first translation step, the compiler thus converts the OpenMP application to an SPMD form, interprets the OpenMP directives and performs the requisite partitioning. After work partitioning, the compiler constructs the set of *shared* data used in the program, using an algorithm described in previous work [8]. Additionally, at this stage, the compiler also does computation repartitioning. Computation repartitioning, discussed previously in the context of deploying OpenMP on Software DSM systems [8], is also important in translating OpenMP applications to MPI. It transforms patterns where the OpenMP application has loops

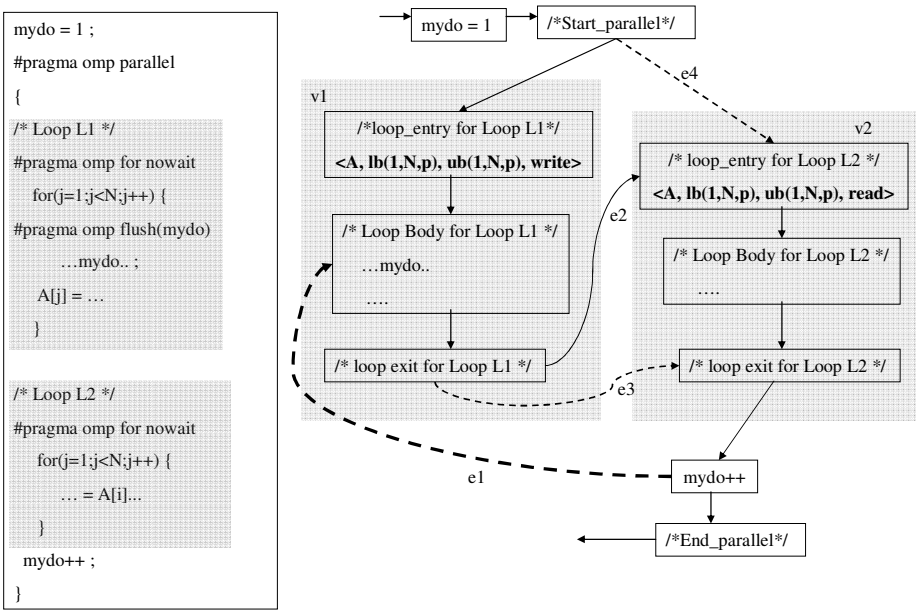


Fig. 1. Creation of the Producer-Consumer Flow Graph: Loop entry vertices $v1$ and $v2$ are annotated with Regular Section Descriptors summarizing accesses to shared array A. Edge $e1$ is added since the flush directive asserts that writes to $mydo$ by one thread in loop $L2$ will be visible to reads of $mydo$ by other threads in loop $L1$. The nowait clause in loop $L1$ implies that writes to A by one thread in loop $L1$ are not guaranteed to be visible to other threads in loop $L2$. Therefore, edge $e2$ is deleted and edges $e3$ and $e4$ are added to break the path from the producer descriptor in vertex $v1$ to the consumer descriptor in vertex $v2$.

partitioned in such a way that multi-dimensional arrays are accessed along different subscripts in different loops. Computation repartitioning then maintains data affinity by changing the level at which parallelism is specified and partitions work using the parallelism of the inner loops.

2.2 Generation of MPI Messages using Array Dataflow

In the second step, the compiler inserts MPI calls to communicate 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 [9] and Regular Section Descriptors [10] have been proposed to characterize array accesses. Our compiler constructs bounded regular section descriptors [11] to characterize accesses to shared arrays. The compiler constructs a control flow graph (with each vertex corresponding to a program statement) and records the Regular Section Descriptors (RSDs) by annotating the vertices of the control flow graph. The starting points of OpenMP work-sharing constructs are also recorded by annotating the loop entry vertices. For each shared array accessed within a loop, the loop entry vertex is annotated with an RSD summarizing accesses within the loop. For accesses to shared-arrays outside loops, the corresponding program statement vertex is annotated with an RSD describing the access.

The compiler then uses this annotated control flow graph to create a *producer-consumer flow graph* for the OpenMP application, which is used to resolve the 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. Writes to shared data by one thread are not guaranteed to be visible to other threads till a memory synchronization point is reached. OpenMP loops have a synchronization point only at the end, thus writes to shared data in one iteration on one thread are not guaranteed to be visible to other iterations on other threads. Thus, in creating the producer-consumer flow graph, the compiler removes the loop back-edge for OpenMP loops to eliminate producer-consumer paths between different iterations of the same loop. Similarly, edges are deleted to account for *nowait* clauses. For explicit synchronization clauses (such as $flush$), a producer-consumer edge is added. This is illustrated by the example in Figure 1. This step is not the same as the insertion of fences [12, 13] discussed in the context of shared-memory applications. Our objective is to ensure that producer-consumer relationships in the producer-consumer flow graph conform to the OpenMP memory consistency model.

Algorithm to Compute Message Sets: The compiler then uses this producer-consumer flow graph for computing message sets to satisfy remote accesses, using the inter-procedural algorithm shown in Figure 2. The objective of computing the message sets is to identify the minimal subset of shared data that a producer p needs to communicate to a *potential* future remote consumer q. The input for this step is the producer-consumer flow graph. The vertex v_{end} marks the end of the program. The graph is traversed starting from annotated Regular Section Descriptor nodes that specify writes to shared-arrays within OpenMP loops. For each shared array written to inside a work-sharing construct, the set of reachable uses is computed. For each reachable use, the corresponding message set is computed by an intersection operation of the regular section descriptors of the current array-write and the reachable use and by subtracting from this the set of intervening productions and consumptions for the array. The computed message sets are communicated using non-blocking *send/receive* and blocking *wait* calls. For affine accesses, the algorithm produces affine expressions for UW_{ij}. If the accesses are regular but not affine, the compiler inserts code to evaluate the symbolic expressions for UW_{ij} at runtime. In case of irregular accesses, the compiler may not be able to precisely determine the sets $PROD_A$ and $NEED_A$. The compiler then conservatively treats these to be the entire array A. Methods for further refining this conservative estimate are discussed in Section 3. Next, the compiler inserts messages for the communication of scalars. A scalar produced inside a parallel region needs

Algorithm ***COMPUTE_MSG_SETS_FOR_PROGRAM***

Input :
The producer-consumer flow graph for the program ,
N - the number of participating processes.
Output :
UW_{ij} - The set of data elements that must be communicated
from process i to process j at the end of work sharing construct W
for each work-sharing construct in the program and $\forall i \forall j, 1 \leq i, j \leq N$.
Operations : We define the following operations -
REACH-ANY-USE(v_1, v_2, A) returns the set of RSD nodes, encountered along *any* (interprocedural) path between v_1 and v_2 (excluding v_1 and v_2), that describe reads to the array A.
REACH-ALL-USE(v_1, v_2, A) returns the set of RSD nodes, encountered along *all* (interprocedural) paths between v_1 and v_2 (excluding v_1 and v_2), that describe reads to the array A.
REACH-ALL-DEF(v_1, v_2, A) returns the set of RSD nodes, encountered along *all* (interprocedural) paths between v_1 and v_2 (excluding v_1 and v_2), that describe writes to the array A.
SEC-READ(v, A) returns the RSD for reads to array A contained in the annotation of node v.
SEC-WRITE(v, A) returns the RSD for writes for array A contained in the annotation of node v.
EVAL(e,p) returns the accesses of RSD e made by process p.

Start ***COMPUTE_MSG_SETS_FOR_PROGRAM***
do for each OpenMP work-sharing construct W
Compute S_W = {Set of shared-arrays written within W}
Initialize $UW_{ij} = \phi, 1 \leq i, j \leq N$
do for each array A, $A \in S_W$,
Let v_{AW} be the entry node for W
Compute $PROD_A$ = SEC-WRITE(v_{AW}, A) //Array elements produced (written) in W
Compute U_A = REACH-ANY-USE(v_{AW},v_{end},A)
Set $WR_A = \phi, UA_A = \phi$
do for each RSD node $v_R \in U_A$,
Compute WR_A = REACH-ALL-DEF(v_{AW}, v_R,A) //Set of intervening producers
Compute UA_A = REACH-ALL-USE(v_{AW}, v_R,A) //Set of intervening consumers
Compute $NEED_A = S1_A - (S2_A \cup S3_A)$
where $S1_A$ = SEC-READ(v_R) , //Array elements consumed in future read
$S2_A = \cup_{v \in WR_A}$ SEC-WRITE(v,A) //Array elements produced by intervening writes
$S3_A = \cup_{v \in UA_A}$ SEC-READ(v,A) //Array elements consumed by intervening reads
Compute $UW_{ij} = UW_{ij} \cup \{EVAL(PROD_A, i) \cap EVAL(NEED_A, j)\}, 1 \leq i, j \leq N, i \neq j$
enddo
enddo
enddo
End ***COMPUTE_MSG_SETS_FOR_PROGRAM***

Fig. 2. Algorithm to Compute Message Sets for Resolving Remote Accesses

to be communicated in two cases - when there is an explicit synchronization directive (such as a flush) after the scalar is written or if the write to the scalar is not part of all paths in the parallel region. In these cases, the producer-consumer flow graph is traversed starting from the write to the scalar and if there are reachable uses, a message is inserted to communicate the scalar.

Techniques for generating communication using RSDs have been proposed in the context of HPF compilers [14, 15]. These techniques have presented communication generation in the context of user specified data-distribution and the owner-computes rule for computation partitioning. Our technique differs in two ways. Firstly, instead of data distribution and owner-computes, our scheme is based on partial replication of shared data and user-specified parallelism (with static block scheduling of parallel loops). Therefore, our compiler looks ahead from every production point of shared data to identify potential future consumers and

formulates the communication accordingly. Secondly, serial regions are redundantly executed in our scheme and thus our compiler needs to generate communication only for data produced in the parallel regions.

3 Translation of Irregular Accesses

A major challenge in translating OpenMP applications directly to message passing is the handling of irregular accesses. Irregular accesses are the reads and writes that cannot be analyzed precisely at compile-time. Existing techniques for dealing with irregular accesses have relied on runtime methods [16, 17]. In this section, we present a technique for handling irregular accesses to shared arrays at compile-time. Section 5 will show that this technique performs well in practice.

To handle such irregular accesses, we make use of the property of *monotonicity*. Monotonicity states $\forall i, \forall j, i \leq j \Leftrightarrow A[i] \leq A[j]$. Testing for monotonicity in irregular accesses has been discussed in the context of parallelizing compilers [18, 19] for analyzing properties of index arrays. We have manually applied these techniques to our benchmarks. We now separately consider irregular reads and irregular writes.

3.1 Irregular Reads

Irregular reads refer to the case where the compiler is unable to exactly determine which process reads which element of an array. In the baseline case, the compiler conservatively overestimates irregular accesses to an array as accesses to the entire array and processes inter-communicate all their preceding writes to the array. This can be done efficiently in our translation scheme since the entire shared data space is allocated on all processes and thus, place-holders already exist on potential consumers to which producers can conservatively forward data. This is different from common HPF implementations, where temporary storage has to be allocated for non-owned shared data, potentially introducing overheads.

However, conservatively forwarding all writes may result in redundant communication. So, as a compile-time optimization, the compiler checks if the future irregular read has a monotonicity property and accordingly refines the accesses. Consider a one-dimensional array A. In a loop, process p writes elements A[l_p] to A[u_p]. In a future parallel loop, there is an irregular read of the form ...=A[B[i]] where i is the loop index. Say, process q executes iterations l_q through u_q of that loop. Then, the compiler traces the evolution of array B to check if it is monotonic. If B can be proved to be monotonic, it suffices to send the subscripts in the range $[l_p, u_p] \cap [B[l_q], B[u_q]]$ from p to q.

3.2 Irregular Writes

Irregular writes refer to the case where it is not possible to exactly determine which process writes which element of an array. In such a case it is generally incorrect to communicate an overestimated range of elements. However, it may be possible to use the monotonicity property to refine the analysis of irregular writes. As an example, consider an array write of the form A[B[i]]=..., where i is the loop index. Process p executes iterations l_p through u_p and process q executes iterations l_q through u_q for the loop. Since parallel loops are statically scheduled with block scheduling, the two iteration spaces cover disjoint intervals, that is, $[l_p, u_p] \cap [l_q, u_q] = \phi$. If the indirection array B can be proved to be monotonic, then we can say that p and q write to the disjoint intervals $[B[l_p], B[u_p]]$ and $[B[l_q], B[u_q]]$. We can now refine the Regular Section Description for the write to array A using these bounds. For the applications that we have encountered, all instances of irregular writes have been analyzable using the monotonicity property.

However, there may be applications where this compile-time scheme is not applicable, either because the indirection function is not provably monotonic or because the irregular write is not in the form of *ARRAY[indirection function]*, In such cases, we use a runtime mechanism as a fallback. In this mechanism, the compiler inserts code to record the writes at runtime. The inserted code allocates a buffer on each process and puts in, at runtime, (location,value) pairs for the elements that are written. At the end of the loop that contains the irregular write, processes intercommunicate this buffer.

4 Optimizations Using Collective Communications

In the previous two sections, we have described techniques for the baseline translation of OpenMP applications to MPI and for handling irregular accesses, respectively. In this section, we describe two additional compile-time optimizations which have been found to improve performance considerably. These are motivated by the observation that most MPI implementations optimize collective communication routines such as MPI_Reduce and MPI_Alltoall. While a detailed evaluation of the impact of each optimization is beyond the scope of this paper, Section 5 indicates the applicability of these optimization techniques for our benchmarks.

4.1 Recognition of Transformed Reduction Idioms

The objective of this optimization is to recognize that the programmer is using a series of OpenMP constructs to perform a reduction operation and to substitute it with a MPI reduction operation. Recognition of reduction

idioms have been discussed in the context of parallelizing compilers [20]. We apply this technique in a way that is the converse of the transformations done by parallelizing compilers. OpenMP does have a reduction clause which can be specified for OpenMP loops. However, this is most commonly used for scalar reductions. Array reduction and other complex forms of reduction operations are often implemented using an OpenMP *for* loop, followed by an OpenMP *critical* section that updates global memory. Instead of a critical section, a parallel loop may also be used where threads update disjoint parts of the shared array in each iteration. In the baseline translation from OpenMP to MPI, an OpenMP critical section is implemented as a section which processes enter in order. Each process leaving the critical section communicates its updates of shared data to the next process entering the critical section. The last process leaving the critical section communicates its updates to all other processes. However, this baseline translation scheme has a large performance overhead, which grows linearly with the number of processes involved. This overhead can be reduced considerably if the compiler recognizes that this pattern represents a transformed reduction.

The basis of recognizing such patterns is to first identify critical sections (or parallel loops) where shared arrays are being updated using commutative operations with private arrays, then verify that these private arrays are produced in an OpenMP loop immediately preceding the critical section. In such a case, we translate the array reduction directly using MPI_Allreduce functions.

4.2 Optimization of Sends and Receives for Shared Data

The global dataflow analysis, described in Section 2, identifies the production and consumption points of the message sets. To overlap computation with communication, messages are initiated early using *non-blocking sends/receives* and completed just before the consumption point at the receiver with a *wait* operation. A similar scheme has been used by some HPF compilers [15]. This scheme has limited benefit when the distance between the production and consumption points is insufficient. In such a case, if the producer-consumer relationships are of the many-to-many type (as computed by the algorithm in Figure 2), our technique recasts the sends and receives to collective communications using the MPI_Alltoallv function.

5 Performance Evaluation

We have implemented the translation steps, described in Section 2, in the Cetus compiler infrastructure [6]. We then manually applied the techniques described in Sections 3 and 4 to seven representative OpenMP

applications. As our benchmarks, we have selected five out of the eight benchmarks from the NAS Parallel Benchmarks suite [21, 22], namely CG, IS, EP, FT and LU; and two benchmarks from the SPEC OMPM2001 suite [23], namely ART and EQUAKE. Our compiler infrastructure handles C programs. Thus for SPEC OMPM2001 we have used two out of the three available C benchmarks and for the NAS benchmarks, we have used the NPB-2.3 OpenMP C versions created by the Real World Computing Partnership (http://phase.hpcc.jp/Omni/benchmarks/NPB/). In this section, we evaluate the performance of these translated OpenMP versions (henceforth referred to simply as the OpenMP versions). For our experiments, we have selected two different platforms – (1) a cluster of sixteen PIII 800 MHz Linux nodes, with 256 MB memory per node, connected by a commodity 100 Mbps Ethernet network and (2) 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. The MPI libraries used for these platforms is MPICH version 1.2.5 on Linux and IBM MPI libraries on the IBM SP2. The back-end compilers are gcc-3.3 on Linux and xlc version 6 on IBM, both at optimization level O3.

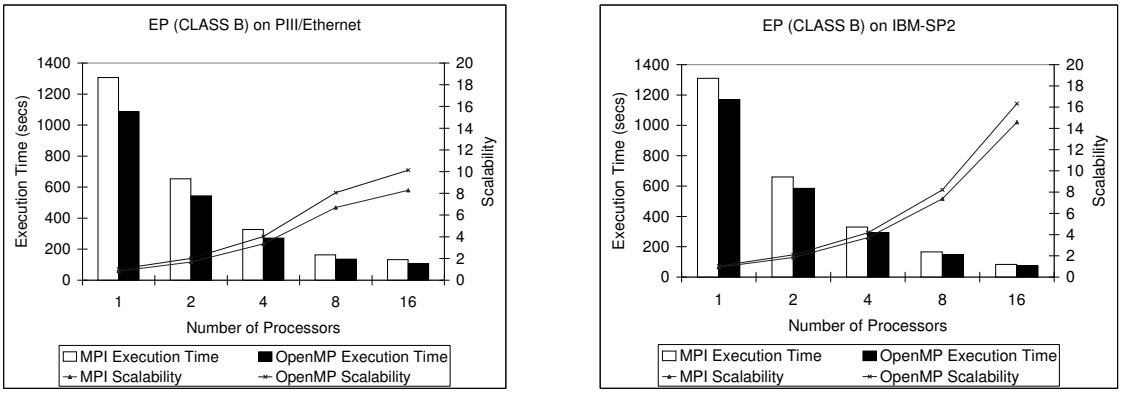
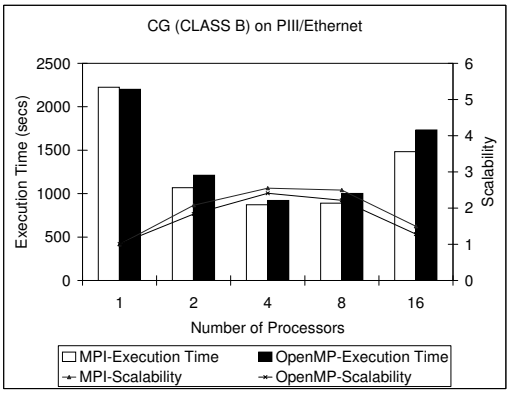


Fig. 3. Performance of EP (CLASS B).

The main objective of this performance evaluation is to compare the performance of the OpenMP versions with the performance of their hand-coded MPI counterparts. The hand-coded MPI versions for CG, IS, EP, FT and LU are part of the NAS Benchmarks NPB2.4 suite. They have been used to evaluate MPI implementations on several platforms and represent an expected upper bound on performance. The SPEC OMPM2001 benchmarks EQUAKE and ART do not have any such existing MPI versions. We created the MPI versions, using reasonable programming efforts.



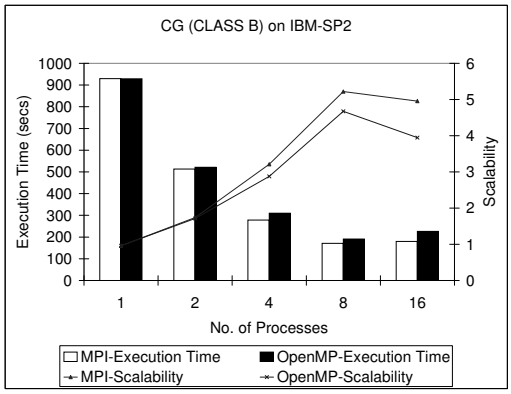
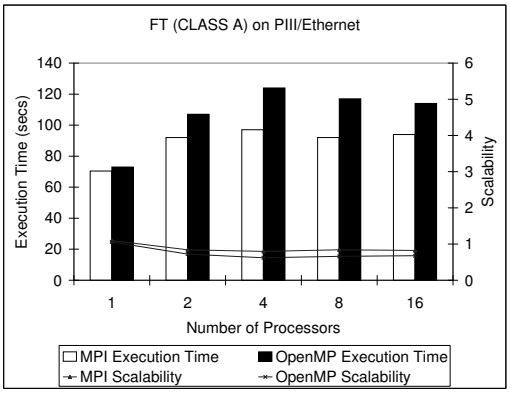


Fig. 4. Performance of CG (CLASS B).



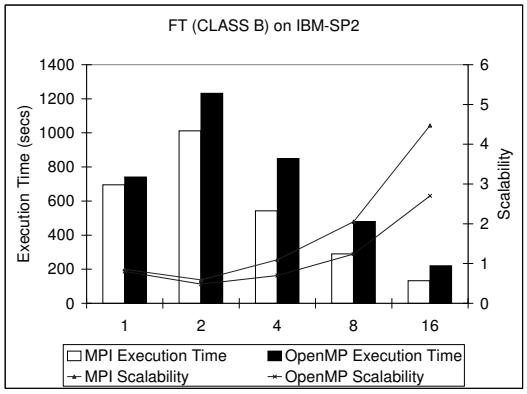


Fig. 5. Performance of FT (CLASS A on Ethernet Cluster and CLASS B on IBM SP2).

Figures 3,4,5,6,7,8 and 9 show the performance of the OpenMP and the MPI versions on the Linux cluster and the IBM SP2. The performance is shown both in terms of the execution time as well as scalability. We define scalability of a version x as $Scalability = Execution\_Time\_of\_serial\_version / Execution\_Time\_of\_version\_x$. The serial versions used are NPB-3.0-SER for the NAS benchmarks and OpenMP versions with the OpenMP directives removed for ART and EQUAKE. Thus we have a common baseline for both OpenMP and MPI versions in measuring scalability. On average, the scalability of our translated OpenMP versions is 12% less and the execution times are 14% more than that of their hand-coded MPI counterparts. Next, we individually examine the performance of each benchmark.

In the case of EP, the MPI version uses several large auxiliary arrays, which are not present in the serial version [22]. Since the OpenMP version is derived from this serial version, its execution time is about 10%

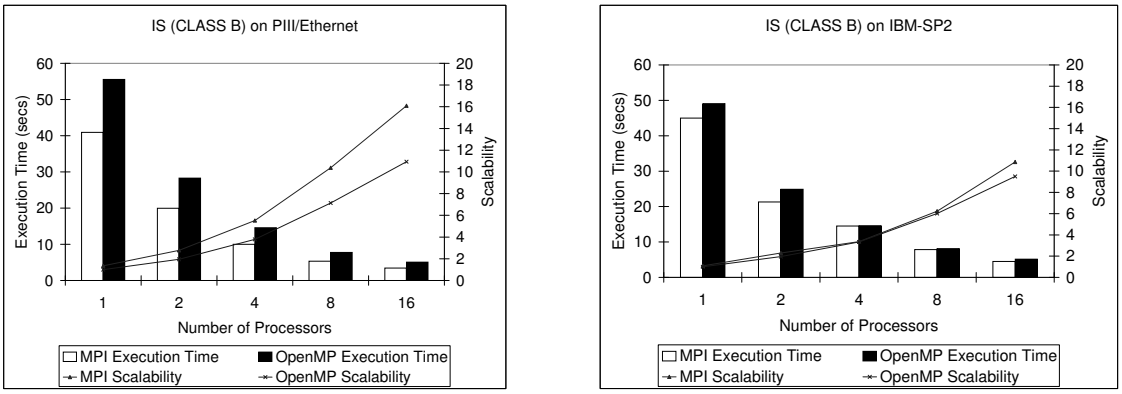


Fig. 6. Performance of IS (CLASS B).

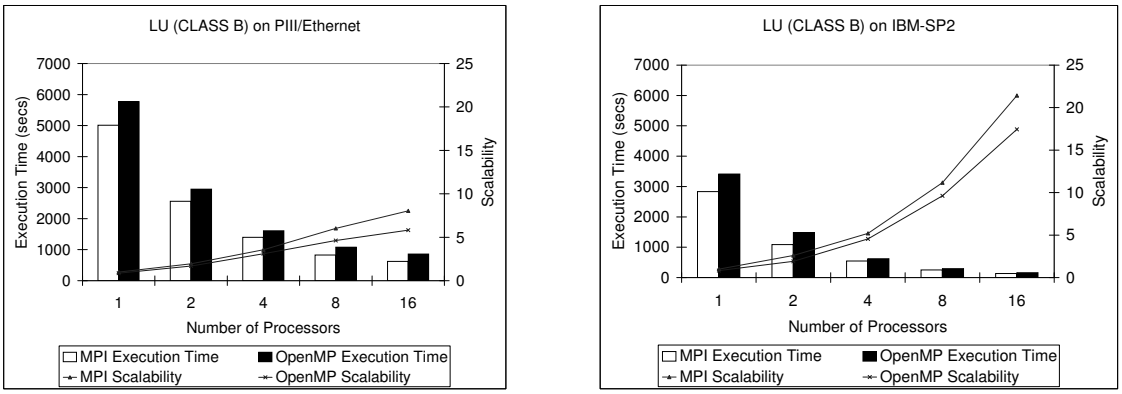


Fig. 7. Performance of LU (CLASS B).

less than the hand-tuned MPI version. The OpenMP to MPI translation of EP benefited from the recognition of an array reduction.

The hand-tuned MPI version of CG benefits from the programmer's knowledge of exact reduction and data exchange patterns in the algorithm. In our translation of the OpenMP version, the message set computation in Figure 2 recognizes the access affinity in the regular accesses for several arrays in the *conj_grad* subroutine. However, there are irregular reads to two arrays. Analysis of the indirection array provides loose bounds for these irregular reads. For these arrays, each process has to broadcast its writes to all other processes. In spite of this difference, our version achieves performance close to that achieved by the hand-coded MPI version.

For FT, the hand-tuned MPI version employs a sophisticated strategy to map the problem into a virtual network topology depending on the problem set size and the number of processes. Furthermore, the resulting transpose communication patterns are also hand-optimized. Owing to these differences, the translated OpenMP version is about 15% slower than its hand-coded MPI counterpart. The dataset used for FT on

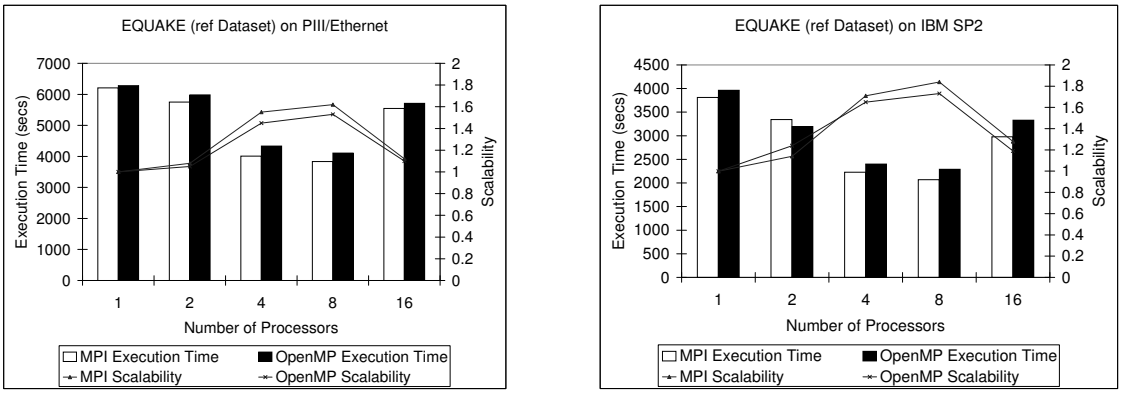


Fig. 8. Performance of EQUAKE (REF Dataset).

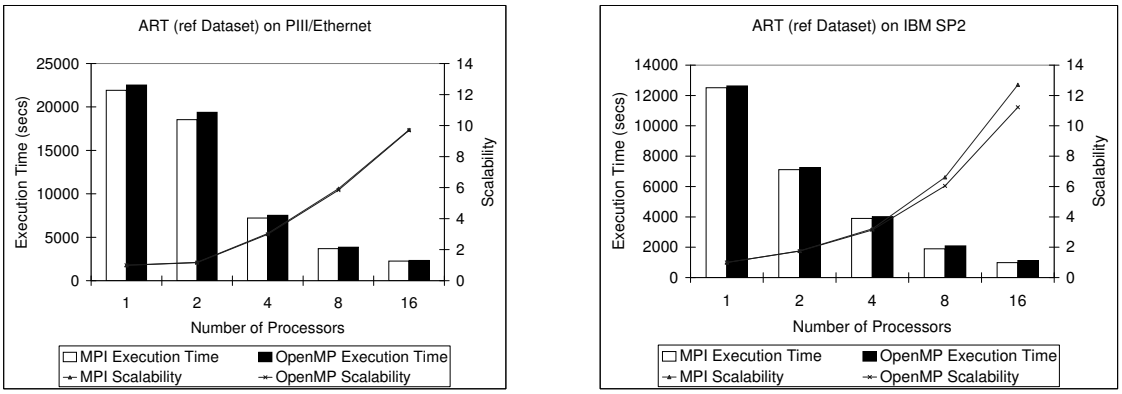


Fig. 9. Performance of ART (REF Dataset).

PIII/Ethernet is CLASS A because the CLASS B problem size was too large (also for the serial and the MPI versions) for nodes with 256 MB main memory.

IS is a NAS benchmark that performs integer sorting. The hand-tuned MPI version of IS uses a *bucket-sort* algorithm. The OpenMP version does not use bucket-sort, in effect performing the ranking using only one *bucket*. The version with buckets has been found to have better cache affinity and the MPI version thus has significantly lower execution times than the OpenMP version (and the serial version without buckets). However, the OpenMP version still achieves scalability to within 14% of the MPI version on both platforms. The translation of IS benefited from the recognition of reduction patterns discussed in Section 4.

LU benefited from the computation repartitioning optimization mentioned in Section 2. The optimized nearest neighbor communication strategy in the hand-tuned MPI version still enables it to achieve better performance and scalability. In LU, we also observed a significantly higher execution time for the OpenMP version on one processor compared to the MPI and serial versions. We attribute a part of this to the difference

in code produced by C and Fortran compilers (the OpenMP version is in C while the MPI and serial versions are in Fortran).

For EQUAKE and ART, the translated versions do not differ significantly from the hand-crafted versions and thus achieve similar performance.

5.1 Comparison with OpenMP deployed using Software Distributed Shared Memory

As discussed in Section 1, recent schemes proposed for the deployment of OpenMP on clusters utilize an underlying layer of Software Distributed Shared Memory (SDSM) [24, 25]. In this section, we compare the performance of the OpenMP versions (translated to MPI using our proposed techniques) with the performance of the OpenMP versions deployed on the TreadMarks SDSM system. For this set of experiments, we have used the sixteen node Linux cluster. Figure 10 shows the performance of the two schemes.

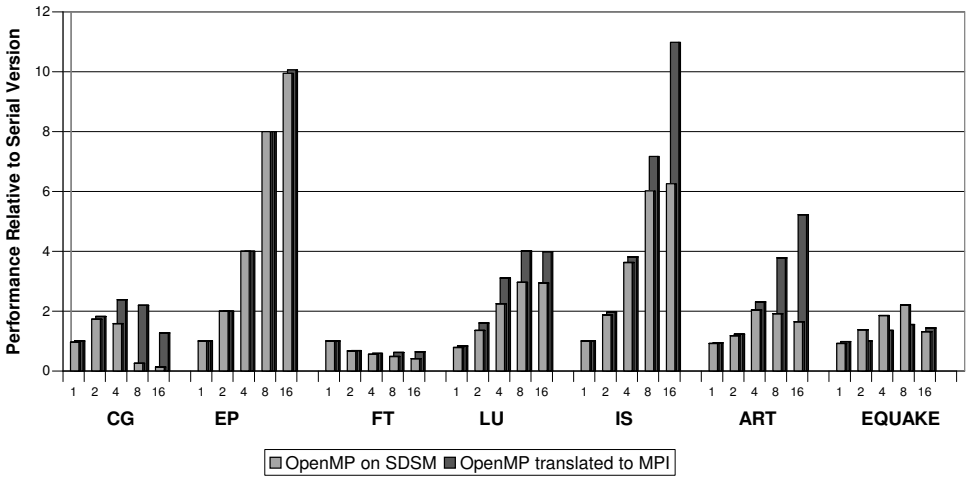


Fig. 10. Performance Comparison of the OpenMP versions translated to MPI with the OpenMP versions deployed on the TreadMarks DSM system. Datasets are *train* for ART and EQUAKE; CLASS A for LU,FT and CLASS B for CG,IS,EP.

Apart from EQUAKE, the versions translated to MPI always perform better compared to the corresponding SDSM versions. EQUAKE has two large arrays that are accessed along different subscripts in the original partitioning scheme. Furthermore, it contains a sparse matrix vector product where it expands the shared data size by allocating work-arrays whose size is proportional to the number of threads. An MPI implementation would need to heavily restructure the existing application to achieve performance. In the SDSM version of EQUAKE, an additional optimization was that iterations for the main loop in the *smvp* (sparse-matrix-vector-product) subroutine were dynamically scheduled, which serialized most accesses and

eliminated communication. Similar performance would be achieved if this loop were serialized for the MPI version as well. For the other benchmarks, the MPI versions mainly profit from the implicit aggregation of messages, which has been cited by others [26] as a principal advantage of message-passing systems.

An important shortcoming of the SDSM versions not evident from the performance comparison, is the constraint on the size of shared data. Each TreadMarks SDSM process creates a *twin* for every page of shared data that it writes to and tracks writes to shared data at runtime by comparing current pages with their original *twins*. Furthermore, memory also needs to be allocated for storing the list of writers to each page of shared data in a program interval. As a result, the maximum size of shared data that can be allocated is constrained to be a fraction of the total available address space. Due to this, one cannot use large datasets such as SPEC's *ref* datasets (which have sizes close to 2 GB) of ART and EQUAKE for the SDSM versions of these benchmarks. By contrast, our translation to MPI does not introduce such size constraints, despite the replication of shared data on all nodes, and we can run the largest problem sizes allowed by the architecture.

6 Related Work

An important contribution towards shared-memory programming support for distributed-memory machines was the development of High Performance Fortran (HPF) [3, 27]. HPF extended Fortran with directives to specify data distribution and data alignment. Some HPF-related compiler efforts are described in [28, 29]. There are important differences between our approach and that taken by HPF. Even though, like OpenMP, HPF provided directives that allowed the user 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 [16].

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 [30], even though data distribution information is not provided by the user. In all our benchmarks, irregular access patterns are handled at compile-time.

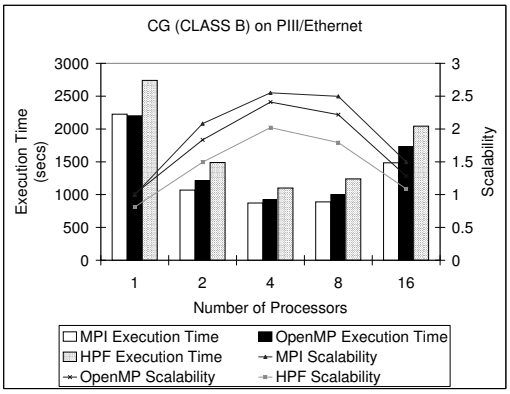


Fig. 11. Performance of CG (CLASS A).

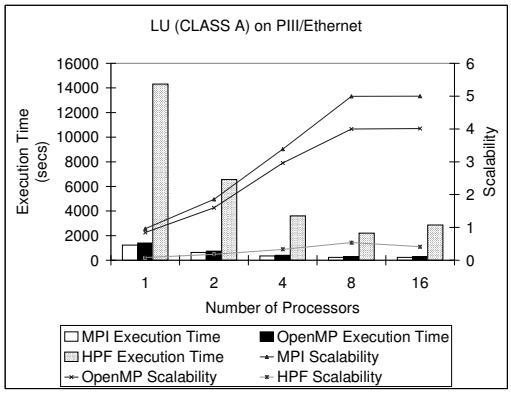


Fig. 12. Performance of LU (CLASS A).

Figures 11 and 12 compares the performance of our programs with those available in HPF form, compiled with the pghpf compiler[1]. Of these programs, we have chosen two extremes: in CG, HPF performs well, while in LU, HPF performs poorly. In both cases, our translation scheme comes close to the MPI performance. In CG our techniques perform 10% better than the HPF versions whereas in LU, our techniques outperform the HPF version 10-fold. The CLASS A dataset has been used for LU, instead of the CLASS B dataset used in comparing OpenMP and MPI, because the available 256 MB memory per node is insufficient to run the HPF version. (We observed the main source of slowdown in the HPF version of LU to be the subroutines *jacld, blts, buts* and *jacu* which have execution times that are more than ten times slower than the serial versions.) These trends are consistent with those reported by the authors of the HPF versions for the NAS benchmarks [31].

Split-C [32] and Unified Parallel C (UPC) [5] are other programming paradigms that have been proposed to ease programming effort by providing a global address space that is logically partitioned between threads. The user specifies affinity between threads and data. In a comparative study for the NAS benchmarks [33], the authors have compared hand optimized UPC versions (hand optimized to privatize local accesses) of the NAS benchmarks with MPI versions. Their results indicate that the UPC versions achieve 50% to 75% the performance of the MPI versions. They find that an important reason for the better scalability of the MPI versions is the optimized implementation of collective operations provided by MPI vendors. Our translation scheme and our optimizations based on collective communication leverages the availability of efficient MPI libraries on multiple HPC platforms.

Recently, several research efforts have suggested the use of OpenMP on clusters. Some of these have proposed language extensions and data distribution directives [34–36] similar to the ones designed for High-

[1] While other HPF compilers may perform better, we believe the pghpf compiler – the only one we could still obtain – to be representative

Performance Fortran (HPF). Others have proposed architectural support and page placement techniques to map data to the most suitable processing nodes [37]. A common scheme suggested for deploying standard OpenMP (without language extensions) on clusters has been to use an underlying Software Distributed Shared Memory (SDSM) system [24, 8]. We have already evaluated the advantage of our approach over deployment using SDSM in Section 5.1.

The tradeoffs between Message-Passing and Shared-Memory programming have been examined often [38–40]. Most of this work has dealt with the criteria for selecting one paradigm over the other in designing multiprocessors and writing parallel programs. Some of this work also identified scenarios where message passing applications may perform better than shared memory applications, even on shared memory systems. In our work we have not compared the applicability of the two models. Rather, we have focussed on the transformation of standard shared-memory programs to a message-passing form.

Some parallelizing compilers have targeted distributed-memory systems and have attempted to directly generate parallel message-passing versions of serial applications [41, 42]. By contrast, our starting point is code parallelized by the programmer. Some research efforts have presented a general framework for matching syntactic reference patterns with appropriate aggregate communication routines [43, 44]. To the best of our knowledge, ours is the only effort that comprehensively addresses the direct, source-level translation of shared-memory programs in standard OpenMP to a portable message-passing form in MPI.

7 Conclusion

We have presented compiler techniques for translating standard OpenMP shared-memory programs into MPI message passing variants, based on a novel model of *partial replication*. The contributions include a new algorithm to compute message sets, techniques for statically handling irregular accesses, and optimizations based on collective communication. Partial replication contrasts with data distribution models, as used in HPF-like languages, in which a single node owns a (partition of a) shared array. Partial replication simplifies the generation of messages, which holds for irregular accesses. Exploiting monotonicity properties of index arrays, our techniques were able to handle all message generation for irregular accesses in our program suite statically. Our methods, although less data scalable than data distribution schemes, can handle much larger data sets than the extensive data replication schemes employed in SDSM systems such as TreadMarks. Our experiments used the full *ref* data sets (up to 2 GB) of the SPEC OMPM2001 benchmarks.

Our measurements show that the presented techniques significantly outperform both HPF programs (using the pghpf compiler) and SDSM programs (using TreadMarks). Perhaps more importantly, they come close to or match hand-tuned MPI programs in all of our measured cases. 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 clusters and highly parallel computer platforms.

References

1. OpenMP Forum, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," Tech. Rep., October 1997.
2. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep. UT-CS-94-230, 1994.
3. High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Tech. Rep. CRPC-TR92225, Houston, Tex., 1993.
4. 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*, vol. 29, no. 2, pp. 18–28, 1996.
5. T. El-Ghazawi, W. Carlson, and J. Draper, "UPC Language Specifications V1.0," Feb. 2001.
6. Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann, "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'03)*. Oct. 2003, pp. 539–553, (Springer-Verlag Lecture Notes in Computer Science).
7. Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister, "A single-program-multiple-data computational model for epex/fortran.," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.
8. Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann, "Optimizing OpenMP programs on Software Distributed Shared Memory Systems," *International Journal of Parallel Programming*, vol. 31, no. 3, pp. 225–249, June 2003.
9. Yunheung Paek, Jay Hoeflinger, and David Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 1, pp. 65–109, 2002.
10. David Callahan and Ken Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *J. Parallel Distrib. Comput.*, vol. 5, no. 5, pp. 517–550, 1988.
11. Paul Havlak and Ken Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, 1991.
12. Dennis Shasha and Marc Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.
13. Arvind Krishnamurthy and Katherine Yelick, "Analyses and optimizations for shared address space programs," *Journal of Parallel and Distributed Computing*, vol. 38, no. 2, pp. 130–144, 1996.
14. K. Kennedy and N. Nedeljković, "Combining dependence and data-flow analyses to optimize communication," in *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, 1995.
15. Manish Gupta, Edith Schonberg, and Harini Srinivasan, "A unified framework for optimizing communication in data-parallel programs:," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 7, pp. 689–704, 1996.
16. R. Das, M. Uysal, J. Saltz, and Yuan-Shin S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, 1994.
17. Reinhard von Hanxleden, Ken Kennedy, Charles H. Koelbel, Raja Das, and Joel H. Saltz, "Compiler analysis for irregular problems in Fortran D," in *1992 Workshop on Languages and Compilers for Parallel Computing*, New Haven, Conn., 1992, number 757, pp. 97–111, Berlin: Springer Verlag.
18. William Blume and Rudolf Eigenmann, "The range test: a dependence test for symbolic, non-linear expressions," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. 1994, pp. 528–537, ACM Press.
19. Yuan Lin and David Padua, "Compiler analysis of irregular memory accesses," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 2000, pp. 157–168, ACM Press.
20. William M. Pottenger and Rudolf Eigenmann, "Idiom recognition in the Polaris Parallelizing Compiler," in *International Conference on Supercomputing*, 1995, pp. 444–448.
21. 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*, vol. 5, no. 3, pp. 63–73, Fall 1991.

22. H. Jin, M. Frumkin, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance.," Tech. Rep. NAS-99-011.
23. 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 the Workshop on OpenMP Applications and Tools (WOMPAT2001), Lecture Notes in Computer Science, 2104*, July 2001, pp. 1–10.
24. Y.C. Hu, H. Lu, A.L. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, pp. 1512–1530, December 2000.
25. Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann, "Supporting realistic OpenMP applications on a commodity cluster of workstations," in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003, Toronto, Canada, June 26-27, 2003. Proceedings Editors: M.J. Voss (Ed.)*, 2003, pp. 170 – 179.
26. Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel, "Quantifying the performance differences between PVM and TreadMarks," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 65–78, 1997.
27. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel Jr., and M.E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.
28. M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo, "An HPF compiler for the IBM SP2," in *Proceedings of Supercomputing '95*, San Diego, CA, 1995.
29. Zeki Bozkus, Larry Meadows, Steven Nakamoto, Vincent Schuster, and Mark Young, "Pghpf-an optimizing high performance fortran compiler for distributed memory machines," *Sci. Program.*, vol. 6, no. 1, pp. 29–40, 1997.
30. Ulrich Kremer, "Automatic data layout for distributed memory machines," Tech. Rep. TR96-261, 14, 1996.
31. M. Frumkin, H. Jin, and J. Yan, "Implementation of NAS Parallel Benchmarks in High Performance Fortran.," Tech. Rep. NAS-98-009.
32. David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick, "Parallel Programming in Split-C," in *Supercomputing*, 1993, pp. 262–273.
33. Tarek El-Ghazawi and Francois Cantonnet, "Upc performance and potential: a npb experimental study," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. 2002, pp. 1–26, IEEE Computer Society Press.
34. 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.
35. 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, October 1998.
36. B. Chapman, P. Mehrotra, and H. Zima, "Enhancing OpenMP with features for Locality Control," Tech. Rep. TR99-02, Inst. for Software Technology and Parallel Systems, U. Vienna, www.par.univie.ac.at., 1999.
37. 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.
38. M. Martonosi and A. Gupta, "Tradeoffs in message passing and shared memory implementations of a standard cell router," in *Proceedings of the 1989 International Conference on Parallel Processing.(IEEE) Volume III*, August 1989, pp. 88–96.
39. Satish Chandra, James R. Larus, and Anne Rogers, "Where is time spent in message-passing and shared-memory programs," in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994, pp. 61–73.
40. A. C. Klaiber and H. M. Levy, "A comparison of message passing and shared memory architectures for data parallel programs," in *Proceedings of the 21th International Symposium on Computer Architecture*, 1994.
41. Jiajing Zhu and Jay 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.
42. Prithviraj Banerjee, John A. Chandy, Manish Gupta, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su, "The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers," in *The First International Workshop on Parallel Processing*, Bangalore, India, Dec. 1994, pp. 322–330.
43. Jingke Li and Marina Chen, "Generating explicit communication from shared-memory program references," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. 1990, pp. 865–876, IEEE Computer Society.
44. J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 3, pp. 361–376, 1991.