Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems *

Ayon Basumallik Rudolf Eigenmann

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

Abstract

In prior work, we have proposed techniques to extend the ease of shared-memory parallel programming to distributed-memory platforms by automatic translation of OpenMP programs to MPI. In the case of irregular applications, the performance of this translation scheme is limited by the fact that accesses to shared-data cannot be accurately resolved at compile-time. Additionally, irregular applications with high communication to computation ratios pose challenges even for direct implementation on message passing systems. In this paper, we present combined compile-time/run-time techniques for optimizing irregular shared-memory applications on message passing systems in the context of automatic translation from OpenMP to MPI. Our transformations enable computation-communication overlap by restructuring irregular parallel loops. The compiler creates inspectors to analyze actual data access patterns for irregular accesses at runtime. This analysis is combined with the compile-time analysis of regular data accesses to determine which iterations of irregular loops access non-local data. The iterations are then reordered to enable computation-communication overlap. In the case where the irregular access occurs inside nested loops, the loop nest is restructured. We evaluate our techniques by translating OpenMP versions of three benchmarks from two important classes of irregular applications - sparse matrix computations and molecular dynamics. We find that for these applications, on sixteen nodes, versions employing computation-communication overlap are almost twice as fast as baseline OpenMP-to-MPI versions, almost 30% faster than inspector-only versions, almost 25%faster than hand-coded versions on two applications and about 9% slower on the third.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Concurrent, distributed, and parallel languages; D.3.2 [*Programming Languages*]: Compilers

General Terms Algorithms, Languages, Performance

Keywords Compiler Techniques, OpenMP, MPI, Performance, Iteration Reordering, Computation-communication overlap.

* 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.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

1. Introduction

OpenMP [15] 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 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 prior work [5], we have proposed techniques to extend the ease of OpenMP shared-memory programming to distributed memory systems by automatically translating *standard* OpenMP programs directly to MPI programs. For applications that contain regular data access patterns, our scheme has achieved performance close to that of hand-tuned MPI applications. An important attribute of this scheme is that it partially replicates shared data. While this characteristic tends to limit the scalability of applications that have large memory requirements, it enables optimizations that were not previously possible.

Irregular applications, where the actual pattern of data accesses are not analyzable at compile-time, are often difficult to parallelize. Converting irregular, serial applications directly to parallel message passing form poses an even greater challenge. Along with parallelization, programmers must perform data-to-process mapping and judiciously insert messages to satisfy remote accesses, even though the actual access patterns are not known at compile-time. This paper aims at making easier such programming efforts by introducing compiler techniques that directly translate irregular OpenMP applications to MPI.

The techniques proposed by the OpenMP-to-MPI translation scheme presented in previous work [5] (henceforth referred to as baseline translation scheme), relied on deducing certain properties (such as monotonicity) of irregular accesses at compile-time. In this paper, we present a more general technique, which is applicable even if the irregular accesses do not exhibit such properties. One observation is that irregular applications are often iterative, whereby the same access pattern (known only at run-time), is repeated over several iterations. The key idea of our compile-time/runtime approach is to overlap computation and communication by restructuring the iteration spaces of irregular loops. This is done in a way that leads to computation on locally available data first, while simultaneously communicating remote data.

Our method makes use of runtime analysis, or *inspection*, techniques. A key insight is that inspecting irregular accesses not only resolves them accurately but also can analyze the precise mapping of accesses to iterations. On each process, this information allows us to partition irregular loop iterations based on the location of the shared data accessed. Computation on local data partitions will then be performed first, overlapped by communication of remote partitions. Next, computation on the closest remote partition will be performed, and so on. When irregular accesses occur within nested loops, the loop may need to be restructured to derive the maximum possible overlap of computation and communication. This paper also describes such restructuring techniques.

Similar issues were considered by proposals to extend High Performance Fortran [20] to handle irregular applications [12]. However, there are several key differences. At a high level, HPF considered the use of data distribution directives, whereas such information is not given explicitly in OpenMP programs. Also, most HPF execution models followed an *owner computes* rule, necessitating inspection for correct execution when the irregular access occurs on the left-hand side of expressions. In our case, inspection is done as an optimization. Our scheme does not have *owners* of shared data. Furthermore, our partial replication scheme allows shared data to be forwarded conservatively in our baseline translation scheme, without allocating extra storage. It also means that the runtime analysis in our case must resolve accesses in terms of producer-consumer relationships derived by the OpenMP-to-MPI compiler whereas in HPF, inspectors must resolve irregular accesses in the context of the specified data distribution. Perhaps most importantly, previous approaches to optimizing irregular accesses on distributed memory machines have not considered loop restructuring for achieving computation-communication overlap.

Restructuring programs to identify potential overlap of computation with communication and inserting non-blocking message passing calls does not yet guarantee improved performance. Current message passing libraries may not ensure that the overlap actually happens. To provide this guarantee, we have implemented a runtime scheme with an explicit communication thread. In our implementation, this thread executes on a separate processor on an architecture with dual-cpu nodes.

The proposed new optimization techniques led to improved performance in *all* our irregular programs where the baseline techniques had limited yield. These baseline techniques came close to the performance achieved by hand-tuned MPI programs, in most of our benchmarks [5]. In Section 5 we will evaluate all three irregular applications where this was not the case. They include the SPEC OMPM2001 program Equake, the NAS benchmark CG, and the molecular dynamics application MOLDYN.

This paper makes the following contributions:

- We present compiler techniques for optimizing the translation of irregular OpenMP programs to MPI, based on runtime analysis and iteration-reordering for computation-communication overlap.
- We present runtime techniques for ensuring true computation-communication overlap. They are based on separate threads for computation and communication.
- We evaluate three representative irregular OpenMP programs from two important classes of irregular applications – sparse matrix computations and molecular dynamics. We compare the performance of our new translation scheme with corresponding hand-tuned MPI program versions, with programs versions translated from OpenMP using the baseline OpenMP-to-MPI translation and with versions that perform runtime analysis without reordering iterations.

The rest of the paper is organized as follows. Section 2 recapitulates the main features of the OpenMP to MPI translation scheme on which our work builds. Section 3 presents techniques for inspection and iteration reordering of irregular applications to accomplish computation-communication overlap. Section 4 presents techniques for creating separate threads for computation and communication, ensuring true overlap. Section 5 evaluates the performance of these techniques. Section 6 places this paper in the context of related work. Section 7 presents conclusions.

2. OpenMP to MPI Translation Scheme

In this section, we briefly revisit the OpenMP to MPI translation scheme proposed in previous work [5]. The objective of this translation scheme is to perform a source-to-source translation of a shared-memory OpenMP program to an MPI program. Our translation scheme transforms OpenMP programs to SPMD [9] style message passing programs with the following characteristics.

- The iteration spaces of all OpenMP parallel *for* loops are partitioned between participating processes. The processes redundantly execute serial regions and parallel regions demarcated by *omp master* and *omp single* directives.
- Shared data, is allocated on all processes. There is no concept of an *owner* for shared data. The analysis only considers where shared data is produced and consumed.
- At the end of a parallel construct, each participating process that has produced shared data communicates this data to all other processes that may consume it in the future.

Our OpenMP-to-MPI translator interprets OpenMP directives to (1) partition iterations of parallel loops between processes using block scheduling, and (2) identifies all the shared data in the program [30]. It then builds a producer-consumer flow graph by (a) summarizing array accesses using Regular Section Descriptors (RSD) [19] of the form $< process-rank, read/write, array-name, array-dimension, lowerbound, upperbound, ... >$ and (b) adding and deleting edges to the program's control flow graph to ensure that dataflow relations derived from the graph conform to OpenMP's relaxed memory consistency model. The translator then traverses this producer-consumer flow graph, starting from each point where shared data is produced and derives the requisite communication sets for communicating shared data to potential consumers. These translation steps have been incorporated into the Cetus [28] compiler infrastructure to build an OpenMP-to-MPI compiler. The idea of communicating data sections from producers to future consumers, rather than maintaining fixed data distributions, has also been pursued in the context of non-uniform shared-memory machines [36].

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 [5]

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.

The above code may be called multiple times during the course of the application's execution, and the indirection array C may not change from one call to the next. To deal with this case, an intuitive optimization would be to inspect an execution instance of loop L2 and record the actual elements of array A accessed by each process. This may help reduce the communication volume the next time that loop L2 is executed. Another observation is that there is no intervening computation between loops L1 and L2. Therefore the inter-communication of array A between processes cannot be overlapped with any intervening communication. Such scenarios severely limit the performance of irregular OpenMP applications translated to MPI and pose challenges even in the direct hand-coding of MPI versions of irregular applications. In the next section, we present an inspection and iteration reordering scheme which alleviates such performance bottlenecks.

3. Inspection and Loop Restructuring for Computation-Communication Overlap

In Section 2, we examined a scenario where run-time inspection may help reduce unnecessary communication in translating irregular OpenMP applications to MPI. Continuing with the example code of Section 2, we see how iteration reordering may allow computation-communication overlap. For the code shown, block scheduling for loop L2 results in processor 0 executing the first 10 iterations, with $j = 0,1,2,\ldots,9$. If the array C is C=2,3,5,9,7,1,4,2,3,5,9,8,2,0,7,6,3,4,5,1, then runtime inspection reveals that process 0 needs to receive A[5],A[9] and A[7] from processor 1 (and not all 5 elements A[5] through A[9]). Now, if process 0 reorders its iterations and the new iteration order for loop L2 on process 0 is j=0,1,5,6,7,8,2,3,4,9 instead of j=0,1..9, then, in the first six iterations, process 0 will access locally available elements. This computation can overlap in time with the receipt of A[5],A[9] and A[7] from process 1. Here, inspection helps to reduce redundant communication and also reveals which iterations access which data elements. Using this information, iteration reordering can be done to overlap communication of remotely produced data with computation that accesses locally available data. Now, consider what happens on process 1. Process 1 performs iterations j=10,11,...,19 of loop L2. Thus, it needs to receive all five elements A[0] through A[4] from process 0. Thus, inspection does not reduce the amount of data process 0 must send to process 1. However, inspection does reveal that if process 1 reorders its iterations of loop L2 to be j=10,11,14,15,18,12,13,16,17,19 then it can execute the first five iterations with locally available data, while it receives the elements A[0] through A[4] from process 0. This is an example where inspection does not reduce communication but it still enables simple iteration re-ordering that exposes available opportunity for computation-communication overlap.

The above iteration re-ordering scheme may not yet expose the maximum available opportunity for computation-communication overlap. Consider a common sparse matrix-vector product shown in Figure 1, 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.

```
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 1. Sparse Matrix-Vector Multiplication Kernel

```
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 2. 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 3. Restructuring of Sparse Matrix-Vector Multiplication Loop

To expose the maximum available opportunity for computation-communication overlap, the loop L2 in Figure 1 is restructured to the form shown in Figure 2. 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 2 to loop L3 in Figure 3. Iterations of loop L3 can now be re-ordered to achieve maximum overlap of computation and communication in our sparse matrix-vector multiplication example.

The OpenMP-to-MPI compiler, in coordination with run-time inspection must affect three transformations of the parallel loops that contain irregular accesses to achieve the described computation-communication overlap. The goal of these transformations is to "conceptually coalesce" the loop nest, followed by the techniques described in the previous paragraph. The three transformations are –

(1) Conversion of loop nests containing irregular accesses to perfectly nested loops.

(2) Inspection of irregular accesses, restructuring of n-dimensional loop nests to one-dimensional loops and reordering of a one-dimensional loop on the basis of inspected data accesses.

(3) Scheduling of overlapping communication with the reordered computation.

Algorithm *create_inspector_executor*
Input :
1. A perfectly nested loop L containing irregular accesses of shared arrays.
Output :
1. An inspector version L' of loop L.
2. An executor version L" of loop L.
3. Linked list of runtime inspection results for each array *a* accessed in loop L. Each inspection structure instance has the form
T_a :{$i_1,...,i_n$: the index vector of the loop nest L,
$x_1,...,x_k$:values of all irregular subscripts for the accesses to array *a*.
}
Start *create_inspector_executor*
1. Set iteration space of L' identical to iteration space of L.
2. Set loop body of L' identical to loop body of L.
3. *do* for each array *a* accessed irregularly in L
4. In the loop body of L', add statements to
5. (a) Allocate a new instance of T_a
6. (b) Assign $T_a.i_1 \leftarrow i1,...,T_a.i_n \leftarrow i_n$
7. (c) Increment $length(T_a)$
8. (d) do for each irregular subscript idx_k of the array *a*
9. Assign $T_a.x_k \leftarrow idx_k$
10. end do
11. (e) Call the runtime library for iteration reordering (algorithm shown in Figure 5)
12. *end do*
13. Create loop L" with single-dimension iteration space $[1, length(T_a)]$
14. Set loop body of L" identical to loop body of L.
15. In the loop body of L", substitute
16. (a) index variables $i_1,...,i_I$ with $T_a.i_1,...,T_a.i_I$, and
17. (b) array subscripts $x_1,...,x_j$ with $T_a.x_1,...,T_a.x_j$.
End *create_inspector_executor*

Figure 4. Compile-time Algorithm to create inspector and executor versions of a loop containing irregular accesses

(I) Conversion of Loop Nests with Irregular Accesses to Perfectly Nested Form This step is a compile-time transformation. The goal of this step is twofold -
(1) convert the irregular parallel loop into multiple loops such that in each loop nest, there is either only a single statement that contains irregular accesses or if there are multiple statements containing irregular accesses, they are at the same loop depth; and
(2) nested loops containing irregular accesses are perfectly nested up to the depth where the irregular access occurs. Several techniques are available for converting imperfectly nested loops to perfectly nested loops [32]. We found that array privatization [35] followed by loop distribution [26] is sufficient to achieve the above two conditions for all parallel irregular loops that we encountered. In our example, this step would convert loop L2 in Figure 1 to loops L2-1 and L2-2 in Figure 2. Additionally, at this step, the compiler verifies the legality of restructuring the loop and re-ordering the loop iterations. Restructuring and reordering are legal only if there are no loop-carried dependencies in the nested loop. For the irregular applications we encountered, recognition of reductions followed by simple dependence tests was sufficient to prove the absence of any loop-carried dependencies.

(II) Inspection, Loop Re-structuring and Iteration Reordering The second step is a combined compile-time/run-time transformation. The input to this step is a set of perfectly nested loops that contain the irregular accesses. From these loops, at compile-time, the compiler generates inspector and executor loops using the algorithm presented in Figure 4. To create the inspector, the compiler inserts code in the original loop to record the values of loop indices and irregular subscripts. To create the executor loop, the compiler first coalesces the original nested loops to one-dimensional loops. Note that by one-dimensional loop, we imply that this transformation is aimed only to lower the nest dimension to the extent required to put the irregular access in the outermost loop. (There may be a k-deep loop at a further depth which does not contain irregular accesses. For our purpose, we can consider this pattern as a single statement rather than a loop nest). The inspector uses an inspection structure T to record the values of the array subscripts for irregular accesses as well as the values of the loop indices. Recording array subscripts is essential in mapping iterations to array accesses. Recording values of loop indices is essential for coalescing the nested loop to a one dimensional loop, especially if the loop bounds are not affine expressions of the surrounding loops.

At runtime, the inspector loop creates the inspection results T for each array accessed irregularly and calls a library for reordering iterations of the executor loop and creation of computation and communication sets. The effect of this sequence of calls to the library is shown in the algorithm in Figure 5. A key input for this step is the Regular Section Descriptor P_{an} for the reaching definitions of array *a* from each producer *n* (see Section 2). Our OpenMP-to-

Algorithm *restructure_and_sort*
Input :
1. Linked list of runtime inspection results T_a for each array *a* irregularly accessed in the loop L.
2. N - the total number of processes.
3. pid - the local process rank.
4. P_{a1},..,P_{aN} - the regular section descriptors of reaching definitions for each array *a* irregularly accessed in the loop L. (See Section 2)
Effect :
Creation of
1. Computation blocks S_1,...,S_N (each containing a chunk of executor loop iterations)
2. Communication blocks C_1,...,C_{N-1}
Start *restructure_and_sort*
1. Allocate sets $prod_1,...,prod_{length(T_a)}$ and initialize them to ϕ
2. $\forall i \in [1,N]$ initialize $S_i = \phi, C_i = \phi$
3. *do* $\forall$ iteration *i* of inspector loop L'
4. *do* $\forall$ array *a* irregularly accessed in iteration *i*
5. *do* $\forall$ irregular subscripts $T_a.x_{t_i}$ of access to *a*
6. *if* $T_a.x_{t_i} \in P_{aK}$ then
7. $C_K = C_K \cup a[T_a.x_{t_i}]$
8. $prod_i = prod_i \cup K$
9. *endif*
10. *end do*
11. *end do*
12. *end do*
13. Sort iterations *j* of L" lexicographically on the tuples $< prod_i, T_a.i_{1j},...,T_a.i_{Ij} >$.
14. Partition the sorted iteration space I' of executor loop L" into the computation sets $S_1,...,S_N$ such that
$\forall$ iteration $i, i \in S_k \Rightarrow prod_i \subseteq \{pid,...,(pid+k-1) moduloN\}$
$\wedge k \in prod_i$
End *restructure_and_sort*

Figure 5. Runtime Algorithm for transformations effected by the sequence of calls from the executor to a runtime library that reorders executor loop iterations and produces computation and communication blocks

Algorithm *schedule_comp_comm*
Input :
1. N - the total number of processes
2. Computation blocks S_1,...,S_N
3. Communication blocks C_1,...,C_{N-1}
4. myid - the local process rank

Effect :
1. Scheduling of computation and communication

Start *schedule_comp_comm*
1. begin non-blocking receives of Communication blocks C_1,...,C_{N-1} from processes $(myid+1)moduloN$,...,$(myid+N-1)moduloN$.
2. *for* j = 1 to $N-1$
3. schedule computation block S_j
4. complete non-blocking receive of communication block C_j
5. *end for*
6. schedule computation block S_N

End *schedule_comp_comm*

Figure 6. Runtime Algorithm to Schedule Overlapping Computation and Communication

MPI compiler uses Regular Section Descriptors (RSDs) to summarize production and consumption information for shared arrays. By comparing the inspection structure T_A for accesses to A with the RSDs of the reaching definitions, this algorithm deduces - (1) the source of data required by a particular iteration and (2) the exact array elements required by a particular iteration. This is a key difference between inspection in our scheme and inspection in the context of HPF. In HPF, the inspector resolves the array sources by comparing accesses with the data distribution specified in the program. In our case, sources are resolved in terms of the regular section descriptors constructed by the OpenMP-to-MPI compiler, to create the producer sets $prod_i$ in Figure 5. The iterations of the one-dimensional executor loop created by the compiler are then lexicographically sorted based on these producer sets as well as the values of the index variables $< i_1, ..., i_n >$. Using the producer sets as the primary key results in contiguous blocks of iterations having the same data source. The compiler then forms computation sets S_k using such contiguous blocks of iterations. Using the index variables $< i_1, ..., i_n >$ as a secondary key in the lexicographical sorting maintains, to the extent possible, the cache affinity of any regular array accesses present in the loops.

At this point, we make two observations regarding the overheads of the algorithm described in Figure 5. For each irregular access, the inspector must determine the data source by comparing the inspected value of the subscript with a Regular Section Descriptor(RSD). Since the RSDs are expressed as ranges, this can be done efficiently. The main overhead of this algorithm involves the lexicographical sorting. However, in our execution model, the irregular parallel loop is distributed blockwise between processes [5] and each process has to perform these steps for only the iteration block assigned to it. Thus, the total iteration space that must be inspected and restructured decreases proportionally with increasing number of processes. Thus, we found that the overhead of this algorithm also decreases proportionally with increasing number of processes.

(III) Scheduling of Overlapping Communication with the Reordered Computation The algorithm described in Figure 5 creates computation and communication blocks that are scheduled by the algorithm described in Figure 6. The fact that the iteration space is already lexicographically sorted in terms of the array accesses results in each computation partition S_k having a set of contiguous iterations. This algorithm simply overlaps the communication for a data block with computation on data that is either locally available or has already been received.

4. Ensuring Computation-Communication Overlap

The inspection technique described in the previous section maps iterations to shared data accesses. The iteration reodering scheme then creates computation and communication sets that can be overlapped with each other. In principle, we could overlap computation and communication by using non-blocking send/receive calls in MPI. However, in practice, we find that this does not necessarily guarantee that background MPI communication will progress while the computation runs in the foreground. As far as the semantics of non-blocking calls is concerned, the MPI specification [14] only states that a non-blocking MPI send call will return irrespective of whether a matching MPI receive call has been posted, and a non-blocking MPI receive call will return irrespective of whether a matching MPI send call has been posted. If a process has called MPI_Irecv (non-blocking receive) there is no guarantee that communication will start and progress as soon as another process posts a matching MPI_Isend (non-blocking send) call. In fact, depending upon the MPI implementation and the amount of data being communicated, this point-to-point communication may not progress to completion till the receiver posts a matching MPI_Wait or MPI_Test call. Many current vendor implementations do ensure progress of non-blocking communication in the background. However, target platforms for our translation scheme also include commodity systems such as network of workstations connected by regular ethernet and running publicly available MPI libraries such as MPICH [16] and LAM [7], where background progress of non-blocking communication is not automatically guaranteed. In the experiments discussed in Section 5, our platform is a set of sixteen dual-processor WinterHawk nodes of an IBM SP2 system, connected by a high-performance switch. Current IBM SP switches have a Remote Direct Memory Access (RDMA) capability, whereby non-blocking MPI communication can progress concurrently with computation. However, the switch used on our platform is less recent and lacks RDMA capability.

To address this lack of progress guarantee, our approach is to split the program into separate computation and communication threads. For each dual processor node in our system, one processor is dedicated for running the communication thread. This approach is also advocated in some current message-passing architectures like BlueGene [2, 3], where the recommended use of one processor on a multi-processor node is as a communication co-processor. A communication thread is spawned per process at the very beginning of the program. Each process routes subsequent MPI calls through this communication thread. When a process needs to perform a non-blocking receive operation overlapped with computation, the computation thread queues a special non-blocking receive call with the communication thread. The communication thread actually performs this non-blocking receive using a blocking receive operation to ensure progress. In this way, true overlap of computation and communication is achieved irrespective of whether the underlying MPI implementation guarantees progress for non-blocking operations.

We have implemented a Pthreads-based library of MPI calls for this purpose. The library contains wrapper functions for most basic MPI calls. For example, for send, there are two wrapper functions. One of these is called by the computation thread. It allocates space to hold the parameters for the send call, copies the parameters into this allocated space and signals the communication thread, passing to it pointers to the allocated space and the second wrapper

function. The communication thread executes the second wrapper function, which performs the actual MPI send call.

A novel feature of our system is that, in addition to the basic wrapper functions in this library, the OpenMP-to-MPI compiler also generates specialized "pack-and-send" and "receive-and-unpack" subroutines per instance of overlapping computation and communication where required. Irregular accesses often result in access to non-contiguous array elements for a contiguous set of iterations, even though they access data from the same source process. In such a case, the non-contiguous elements need to be packed into a buffer by the sender before being sent and need to be unpacked at the receiver before being used. By generating the specialized subroutines, the compiler assigns the packing and unpacking tasks to the communication thread and thus, manages to hide the packing and unpacking latencies at run-time. From a performance viewpoint, this generating specialized "pack-and-send" and "receive-and-unpack" subroutines is a more efficient alternative to having generalized wrappers for the MPI functions. It also helps maintain cleaner semantics regarding which non-blocking MPI calls should be actually executed by the communication thread in a blocking manner.

5. Performance Evaluation

To evaluate the effectiveness of our reordering scheme, we have manually applied the transformations described in Section 3 to three representative OpenMP benchmarks from two important classes of irregular applications – sparse matrix computations and molecular dynamics. For sparse matrix computations, we have selected Equake from the SPEC OMPM2001 benchmarks [4] and CG from the NAS Parallel Benchmarks 2.3 suite [25]. For molecular dynamics, we have selected the MOLDYN kernel from CHARMM [6] application. Our compiler infrastructure handles C programs. Thus for SPEC OMPM2001 we have used the only available irregular C benchmark. For the NAS benchmarks, we have used the NPB-2.3 OpenMP C version of CG created by RWCP (http://phase.hpcc.jp/Omni/benchmarks/NPB/).
CG and IS are the two NAS benchmarks that have irregular accesses. In previous work [5] we had already achieved satisfactory scalability for IS. Therefore, for this study, we have selected CG. For each of these applications, we evaluate the effects of our transformation using two metrics – (1) in terms of the reduction in execution time for the whole application and (2) in terms of the actual computation-communication overlap achieved by the reordering.

For evaluating the reduction in execution time for the whole application, we compare the execution times of
(a) A hand-coded MPI version, that does not use computation-communication overlap. In case of CG, this version is the official NPB2.3-MPI CG version. For MOLDYN and Equake, we have created these versions ourselves with modest programming effort. (b) A baseline version translated from OpenMP to MPI [5] using only compile-time analysis. This version does not incorporate any run-time inspection.
(c) A version translated from OpenMP to MPI that uses a run-time inspector, uses non-blocking communication to overlap communication between processes, but does not perform iteration reordering for computation-communication overlap.
(d) A version translated from OpenMP to MPI that uses a run-time inspector and also reorders iterations for computation-communication overlap.

For measuring the actual computation-communication overlap achieved by iteration reordering, we individually measure the actual time spent in sends/receives by the communication thread, the overlapping computation and the actual time spent waiting by the computation thread for the required communication to complete. Figure 7 depicts a typical scenario that may occur in an execution with three involved processes p,q and r.

The total time spent in MPI Send/Recvs by the communication thread of process p is $t_{send-recv} = t_{send} + t_{recv-q} + t_{recv-r}$ and the computation time available for overlap is $t_{comp} = t_{comp-p} + t_{comp-q}$. The actual time spent by the computation thread waiting for communication to complete (which we refer to as the "actual wait time" is $t_{wait} = t_{wait-q} + t_{wait-r}$. We would thus achieve a reduction in execution time if we have $t_{wait} < t_{send-recv}$. Ideally, we should also see $t_{send-recv}$ being equal to $(t_{comp} + t_{wait})$. In practice, we observe $(t_{comp} + t_{wait}) > t_{send-recv}$ and the difference is because of overheads in threading (waking up waiting computation or communication threads) and in copying data between computation and communication threads.

The platform we have used for our experiments is sixteen IBM SP2 WinterHawk-II dual-processor nodes, connected by a high-performance switch. To ensure background communication progress, as discussed in Section 4, we use one processor per node for the communication thread. Our setup is thus more general and can be extended to other commodity platforms such as multiprocessor workstations connected by regular ethernet. For comparison, wherever relevant, we shall refer to the performance achieved by the inspector-only versions where all thirty two processors were used as compute nodes. The MPI library used is the IBM MPI library with xlc version 6 (at optimization level O3) as the back-end compiler. We now examine in detail the performance of Equake, CG and MOLDYN using the two metrics discussed above.

5.1 Performance of *Equake*

Equake is an OpenMP application that is part of the SPEC OMPM2001 benchmark suite. Equake performs a simulation of an earthquake. For our experiment, we have used the *ref* dataset. However, instead of the 3333 timesteps for the dataset, we have performed measurements for 330 timesteps. The most time-consuming part of this application is the *smvp* subroutine, which computes the product of a sparse matrix (which contains the grid coordinates) and a vector (which contains displacements in time). This subroutine is called in every timestep. The sparse matrix-vector multiplication loop is parallelized in OpenMP to accumulate the product in a private copy in each thread, which is then merged by a global reduction.

Figure 8 shows the performance of the hand-coded, baseline, the inspector-only and the inspector-with-reordering versions for Equake. The sparse matrix-vector product in the smvp subroutine accesses the displacement vector irregularly. However, the access pattern remains the same through all timesteps. The vector read in one timestep is produced blockwise by all threads in the previous timestep. The compiler analysis for the OpenMP-to-MPI baseline translation assumes that each thread reads the entire vector and thus inserts an all-to-all communication between processes to communicate the displacement vector. The resultant large volume of communication limits the scalability of this baseline translation. Using an inspector in the "inspector-only" version cuts down the communication requirement by recording the actual accesses pattern for the displacement vector and shows a speedup of close to three times (for 16 processes) over a serial version. The hand-coded version uses inspection of the indirection vector as well to cut down communication, but does not overlap computation with communication. Its performance is better than the baseline and inspector-only versions. But re-ordering iterations and using computation-communication overlapping further reduces execution time by about 32% on 8 nodes and 45% on sixteen nodes compared to the inspector-only version and is faster than the hand-coded version on eight and sixteen processes as well. The increased single-processor execution time for the inspector-only

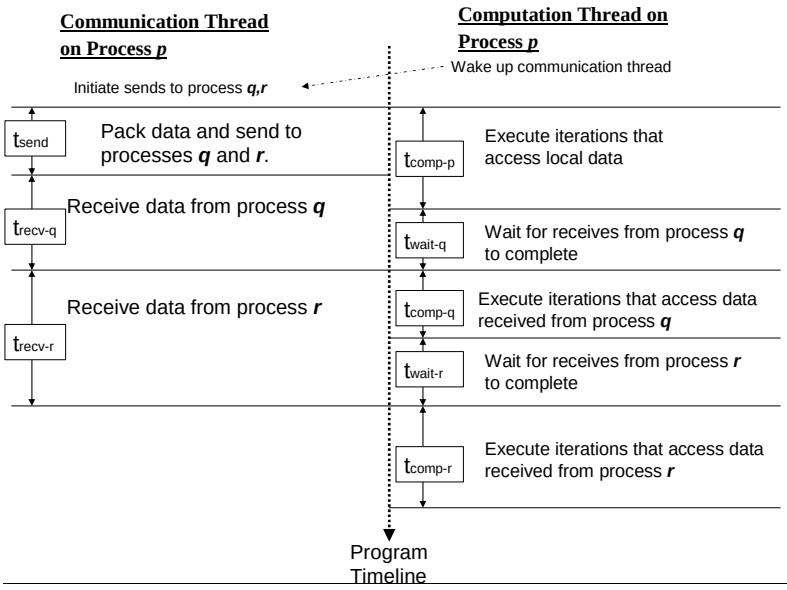

Figure 7. Measurement of Computation-Communication Overlap : A scenario with three processes - *p*,*q* and *r*. All processes send data computed locally to all other processes that need these data in the next step. Measured on process *p*, the time spent in sends/recvs is $(t_{send} + t_{recv-q} + t_{recv-r})$. The computation available for overlap is $(t_{comp-p} + t_{comp-q})$. The actual wait time (time spent by the computation thread on p waiting for sends/receives to complete) is $(t_{wait-q} + t_{wait-r})$.

and the inspector-with-reordering versions is the inspection and reordering overhead. This overhead also reduces in going from one to sixteen processors because the iteration space (that needs to be inspected and reordered) on each process reduces with increasing number of processes. Even if all 32 processors in the 16 nodes are used, the 16 node inspector-with-reordering version is faster than the 32-process hand coded version.

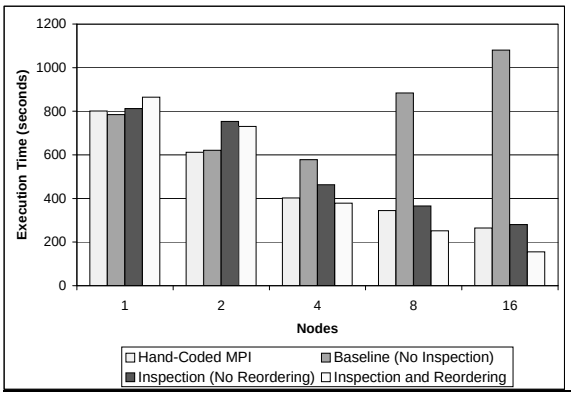

Figure 8. Performance of *Equake*.

Figure 9 shows the computation-communication overlap achieved for Equake. Specifically, we have measured this metric for the matrix-vector part in the *smvp* subroutine. Even after inspection, the communication volume is very high. However, overlap allows us to tolerate the communication latency considerably, cutting down the effective communication latency (t_{wait} as discussed previously) by about 50 % on two processes to 68 % on sixteen processes. This reduction of communication latency of the matrix-vector product, as well as the communication latency reduction for the subsequent accumulation of the result vectors, makes the reordered version almost twice as fast as the inspector-only version on sixteen nodes. An observation form Figure 9 is that the actual-time spent in sends/receives reduces in going from two to sixteen nodes. This trend depends on the actual pattern of communication between processes and the volume of data that needs to be communicated. For CG, the time spent in sends/receives increases in going from two to sixteen nodes.

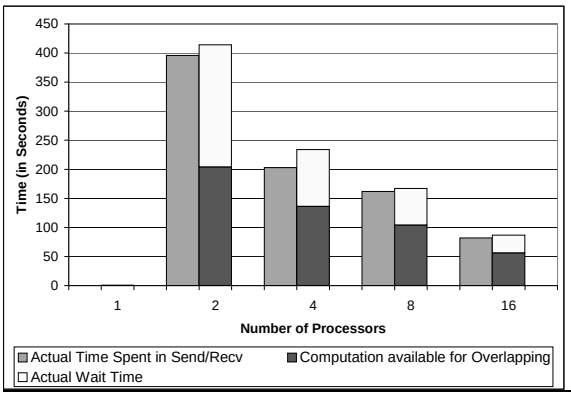

Figure 9. Computation-Communication Overlap in *Equake*.

5.2 Performance of *CG*

CG implements the conjugate-gradient method and is part of the NAS benchmark suite. The most time-consuming part of this application is the *conj_grad* subroutine and a large chunk of this time is spent in computing the product of a sparse matrix *A* and a vector *p*. In the OpenMP matrix-vector multiplication loop, *A* is accessed blockwise and *p* is irregularly read. Figure 10 shows the performance of the baseline, inspector-only and inspector-with-reordering versions.

The baseline translation assumes that the entire vector *p* is read by all processes. As it turns out, this assumption is actually valid. Therefore, the inspector-only version does not reduce any communication time and is not any faster than the baseline version. The inspector-with-reordering version succeeds in tolerating the communication latency and reducing the execution time by almost 22%

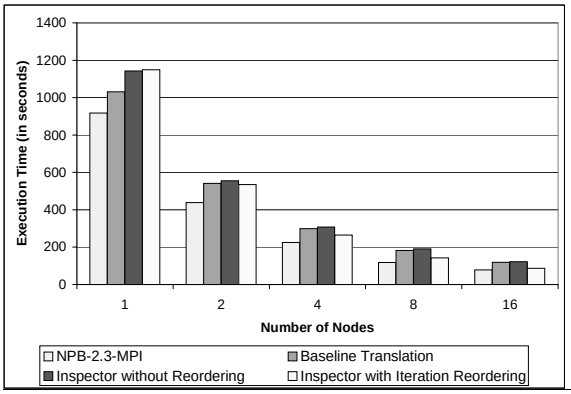

Figure 10. Performance of *CG*.

on 16 nodes compared to the baseline version. Figure 10 also shows the execution of the hand-coded NPB2.3-MPI version of CG. This version employs a sophisticated strategy of logically dividing processes into a 2-D grid, distributing p along columns and replicating it along rows in order to cut down communication overhead. Still, we find that the MPI version created by translation from the comparatively simple OpenMP version is just about 10% slower than the hand-coded MPI version. The CG benchmark performs one iteration for "free" before it starts the timed iterations. The inspector and reordering code executes during this free iteration and the overheads are not directly visible in this performance comparison. However, we separately measured this overhead and found it to be about 3.8% of the complete application's execution time.

Figure 11 shows the computation-communication overlap in CG. The actual time spent in sends/recvs ($t_{send-recv}$) for CG is (for up to 16 nodes) lower than the computation time available for overlapping. Therefore, the communication latency can be almost entirely tolerated except for overheads involved in threading and in moving data between the computation and communication threads. Thus, the actual time for the entire application spent by the computation thread waiting for communication to complete (t_{wait}) is very low (about 2 seconds).

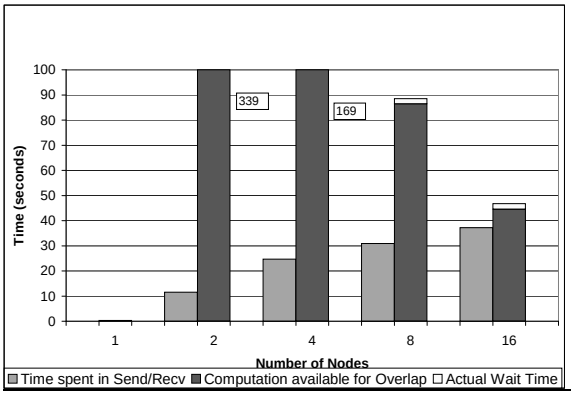

Figure 11. Computation-Communication overlap in *CG*.

5.3 Performance of *MOLDYN*

MOLDYN is a kernel within the CHARMM molecular mechanics application. MOLDYN simulates the interaction between particles by considering only those pairs of particles that are within a cutoff distance from each other. It builds an interaction list of such "neighbors" right at the beginning and computes changes in each particle's position, velocity and energy over certain number of timesteps. After a given number of timesteps, it recomputes the list of neighbors.

The two most time-consuming parts of MOLDYN are (a) computing the list of interacting particles (neighbors) and (b) computing the forces between interacting pairs for all pairs in the interaction list. In creating the OpenMP version of MOLDYN, we followed an SPMD style parallelization approach [22] and were able to effectively parallelize both the force computation and neighbor list creation. Our serial MOLDYN version uses a Recursive Coordinate Bisection partitioner [33] to partition particles prior to building the interaction list. For the OpenMP versions, we have continued to use this partitioner. Irregular accesses occur to the coordinate vectors of the particles in the force computation loop - where each iteration calculates the forces between a pair in the interaction list. The forces are accumulated in a local copy and copies are merged in a subsequent global reduction.

In our experiments, the dataset used for MOLDYN had 23328 particles, the interaction list was computed once at the beginning and then after every 20 timesteps. The simulation was done for a total of 40 timesteps. Figure 12 shows the performance of the hand-coded, baseline, inspector-only and inspector-with-reordering versions of MOLDYN in terms of execution time. The baseline version assumes, for the force computation loop, that each process accesses the entire coordinates vector. It therefore suffers from redundant computation. The hand-coded version inspects the indirection vectors implicitly while the interaction list is being built and is thus more efficient than the inspector-only version. Inspection reduces the volume of communication significantly by eliminating redundant communication and the inspector-only is almost 40% faster than the baseline version. However, iteration reordering and computation-communication overlap tolerates the communication latency even further and that version is faster than both the inspector-only and hand coded versions on four, eight and sixteen nodes.

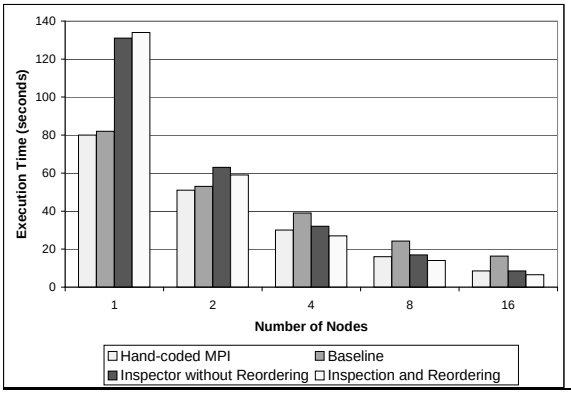

Figure 12. Performance of *MOLDYN*.

Figure 13 shows the computation-communication overlap for the inspector-with-reordering version. On two and four nodes, the available computation is more than the send-recv latency and the communication latency is almost entirely tolerated. For the two node case, there is a slight communication imbalance (for the force calculation, process 1 must receive some coordinates from process 0, but process 0 does not need any coordinates from process 1. Therefore, process 0 completes its force calculation earlier and has to wait a bit in the subsequent global reduction phase, while process 1 completes its force calculation).

Compared to CG and Equake, MOLDYN has a higher overhead associated with inspection of irregular accesses and iteration reordering. For MOLDYN, every time the interaction list is recomputed, inspection of accesses and iteration reordering has to be repeated. However, as has been explained in Section 3, this overhead decreases with increasing number of processes. Despite this over-

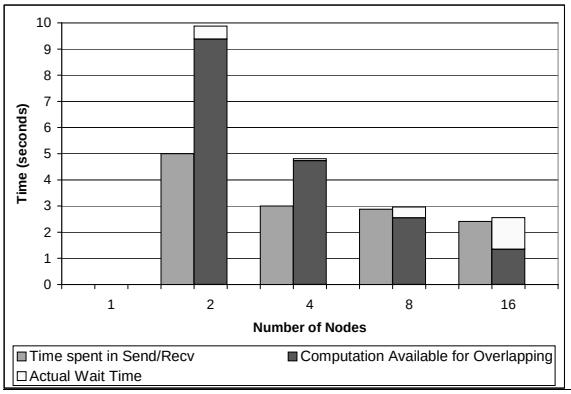

Figure 13. Computation-Communication overlap in *MOLDYN*.

head, the inspector-with-reordering version on 16 nodes shows a speedup of 12.6 relative to the baseline serial version, compared to 9.6 for the inspector-only version and 5.03 for the baseline version on 16 nodes.

6. Related Work

Irregular applications have been studied extensively. There is a large body of work on both compile-time and dynamic schemes [13, 8, 18, 23, 31] which have focussed on improving data locality in irregular applications. While these were important starting points for our work, our focus is on transformations that expose opportunities for computation-communication overlap, rather than data locality.

In the context of irregular parallel applications using HPF, the inspector-executor scheme [12] was proposed to resolve remote accesses. While related approaches proposed communication optimizations [11], none of these approaches examined the restructuring of irregular loops to overlap computation with communication. An instance of loop restructuring within the inspector-executor model was proposed to improve data locality [10].

The notion of tolerating communication latency using computation-communication overlap has been used before, mainly in the context of regular applications. HPF compilers [17] proposed the posting of sends as early as possible and receiving as late as possible in order to overlap with available intervening computation. A key distinction of these with our work is that we overlap communication with *related* computation. Considering the loops L1 and L2 in Figure 1, the simple approach of sending early and receiving late would not achieve any overlap. The vector *p* produced in loop L1 is consumed in loop L2 and there is no intervening computation between L1 and L2 with which the communication of *p* can be overlapped. This pattern is found often. In contrast to HPF compilers, we overlap communication of data used in loop L2 with computation in the same loop.

Some later approaches, in the context of HPF, have suggested the overlapping of computation and communication for a single-loop but these have been compile-time approaches, applicable only for regular loops. Liu and Abdelrahman [1] suggested loop peeling in regular HPF applications to differentiate iterations with local and non-local accesses. They did not consider loop restructuring techniques to maximize overlap opportunities. Ishizaki et al [24] proposed a tiling-based approach for computation-communication overlap in HPF applications. Both these efforts are limited to regular applications. Su and Yelick [34] examined irregular applications in the context of the Titanium [21] language. Their work evaluated message optimizations based on runtime inspection of accesses. They did not propose any loop restructuring to enable computation-communication overlap.

Lu et al [29] proposed compiler support for irregular applications on software distributed shared memory. Their approach was to use the compiler to identify indirection arrays for irregular accesses and to prefetch them, aggregating the irregular accesses. Their approach amortizes the communication latency that would occur in a page-based DSM, but they do not achieve any computation-communication overlap, which could reduce latencies further. Lain and Banerjee [27] have examined a graph-coloring based partitioning approach for iterative irregular applications. Their work is focussed on computation partitioning, rather than any restructuring to expose overlap opportunities in partitions already assigned to a process.

To the best of our knowledge, this paper presents the first approach to optimization shared-memory irregular applications on distributed-memory systems using loop restructuring coupled with run-time inspection to achieve computation-communication overlap.

7. Conclusion

In prior work, we have presented techniques that can transform shared-memory programs, written in standard OpenMP, into MPI message-passing programs. In all but a number of irregular applications, these techniques achieved good performance. In the present paper we have now added techniques that fill this void, leading to good performance in all of our test programs. These include codes from the SPEC OMP applications, NAS benchmarks, and several others.

Our techniques represent a significant step towards increasing the productivity of parallel applications. Writing shared-memory programs is arguably much easier than writing message-passing code. The presented techniques enable programmers to write shared-memory programs and still benefit from low-cost, commodity platforms, such as clusters of PCs. Our model comes at the price of limited data scalability for applications with large memory requirements. The model partially replicates shared data, giving up a benefit that is often associated with distributed-memory platforms: the per-node memory can be smaller than that of a single-processor system for the same application. In return, our model enables new optimizations not previously possible. Hence it trades data scalability against productivity and performance.

Our new optimizations achieve overlap of computation and communication, even in the case where data is produced in one computation loop and consumed in the immediately following loop. It does so by reordering computation in a way that locally available data is used immediately, while remote data is communicated for the next computation step. A combined compile-time/runtime scheme is used to accomplish this. Our runtime techniques also include a method to guarantee that overlap of computation and communication actually happens, which is not the case in many current MPI implementations.

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] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, and J. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.

[3] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. C. nos, L. Ceze, P. Crumley, C. Erway, J. Gagliano, D. Lieber, X. Martorell,

J. E. Moreira, A. Sanomiya, and K. Strauss. An overview of the BlueGene/L system software organization. In *Proceedings of Euro-Par 2003 Conference*, Lecture Notes in Computer Science, Klagenfurt, Austria, August 2003. Springer-Verlag.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001), Lecture Notes in Computer Science, 2104*, pages 1–10, July 2001.

[5] 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.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4:187–217, 1983.

[7] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[8] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 252–262, New York, NY, USA, 1994. ACM Press.

[9] 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.

[10] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*, 1992.

[11] 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.

[12] R. Das, J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Trans. Comput.*, 44(6):737–753, 1995.

[13] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, New York, NY, USA, 1999. ACM Press.

[14] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.

[15] O. Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, Oct. 1997.

[16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[17] 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.

[18] H. Han and C.-W. Tseng. A comparison of locality transformations for irregular codes. In *LCR '00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 70–84, London, UK, 2000. Springer-Verlag.

[19] 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.

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

[21] 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.

[22] D. Hisley, G. Agrawal, P. Satya-narayana, and L. Pollock. Porting and performance evaluation of irregular codes using OpenMP. *Concurrency: Practice and Experience*, 12(12):1241–1259, 2000.

[23] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 127–136, London, UK, 2001. Springer-Verlag.

[24] K. Ishizaki, H. Komatsu, and T. Nakatani. "a loop transformation algorithm for communication overlapping". *International Journal of Parallel Programming*, 28(2):135–154, 2000.

[25] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011.

[26] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 407–416, Washington, DC, USA, 1990. IEEE Computer Society.

[27] A. Lain and P. Banerjee. Techniques to overlap computation and communication in irregular iterative applications. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 236–245, New York, NY, USA, 1994. ACM Press.

[28] S.-I. Lee, T. A. Johnson, and R. 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)*, pages 539–553. (Springer-Verlag Lecture Notes in Computer Science), Oct. 2003.

[29] 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.

[30] 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.

[31] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 192–202, 1999.

[32] R. Sass and M. Mutka. Enabling unimodular transformations. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 753–762, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[33] H. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

[34] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, Denver, Colorado, 2005.

[35] P. Tu and D. Padua. Array privatization for shared and distributed memory machines (extended abstract). *SIGPLAN Not.*, 28(1):64–67, 1993.

[36] A. Yoshida, K. Koshizuka, and H. Kasahara. Data-localization for fortran macro-dataflow computation using partial static task assignment. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 61–68, Philadelphia, Pennsylvania, USA, 1996. ACM Press.