Automatic Scaling of OpenMP Beyond Shared Memory

Okwan Kwon[1], Fahed Jubair[1], Seung-Jai Min[2],
Hansang Bae[1], Rudolf Eigenmann[1] and Samuel Midkiff[1]

[1] Purdue University*
[2] Lawrence Berkeley National Laboratory
kwon7,fjubair@purdue.edu, SJMin@lbl.gov,
baeh,eigenman,smidkiff@purdue.edu

Abstract. OpenMP is an explicit parallel programming model that offers reasonable productivity. Its memory model assumes a shared address space, and hence the direct translation - as done by common OpenMP compilers - requires an underlying shared-memory architecture. Many lab machines include 10s of processors, built from commodity components and thus include distributed address spaces. Despite many efforts to provide higher productivity for these platforms, the most common programming model uses message passing, which is substantially more tedious to program than shared-address-space models. This paper presents a compiler/runtime system that translates OpenMP programs into message passing variants and executes them on clusters up to 64 processors. We build on previous work that provided a proof of concept of such translation. The present paper describes compiler algorithms and runtime techniques that provide the automatic translation of a first class of OpenMP applications: those that exhibit regular write array subscripts and repetitive communication. We evaluate the translator on representative benchmarks of this class and compare their performance against hand-written MPI variants. In all but one case, our translated versions perform close to the hand-written variants.

1 INTRODUCTION

The development of high-productivity programming environments that support the development of efficient programs on distributed-memory architectures is one of the most pressing needs in parallel computing, today. Many of today's parallel computer platforms have a distributed memory architecture, as most likely will future multi-cores.

Despite many approaches [1–4] to provide improved programming models, the state of the art for these platforms is to write explicit message-passing programs, using MPI. This process is tedious, but allows high-performance applications to

* This work was supported, in part, by the National Science Foundation under grants No. 0751153-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

be developed. Because expert software engineers are needed, many parallel computing platforms are inaccessible to the typical programmer. In this paper we describe the first automatic system that translates OpenMP to MPI, and we introduce the key compiler and runtime techniques used in our system. OpenMP has emerged as a standard for programming shared memory applications due to its simplicity. OpenMP programmers often start from serial codes and incrementally insert OpenMP directives to obtain parallel versions. Our aim is to extend this ease of parallel programming beyond single shared-memory machines to small-scale clusters.

Like OpenMP on SDSMs, this approach supports standard OpenMP, without the need for programmers to deal with the distribution of data, as in HPF, UPC, and Co-array Fortran. Previous work [5] has shown that programs could be systematically translated by hand from OpenMP to MPI and achieve good performance. In this work, we aim to automate this translation, using a novel split between the compiler and the runtime system. At an OpenMP synchronization point where coherence across threads is required, the compiler informs the runtime system about array sections written and read before and after the synchronization point, respectively; from this information the runtime system computes the inter-thread overlap between written and read array sections and generates the necessary communication messages. The compiler identifies and utilizes several properties exhibited by many scientific parallel benchmarks for efficient generation of MPI code. Our initial goal, and the focus of this paper, is the class of applications that involve regular array subscripts and repetitive communication patterns. Our translator accepts the OpenMP version 2 used by the benchmarks.

This paper makes the following contributions. We

i. introduce an automatic approach for translating shared-address-space programs, written in OpenMP, into message-passing programs.
ii. introduce a novel compile/runtime approach to generate inter-thread communication messages.
iii. describe the implementation of the translator in the Cetus [6] compiler infrastructure.
iv. present the evaluation of five benchmarks' performance compared to their hand-written MPI counterpart.

We measured the performance on up to 64 processors. Three out of five benchmarks (JACOBI, SPMUL and EP) show comparable performance to the hand-coded MPI counterpart. One benchmark (CG) follows the performance of the hand-coded MPI version up to 16 processors and reaches 50% of the hand-coded MPI performance on 64 processors. The fifth benchmark (FT) has superior performance in the hand-coded MPI version, which we study in detail.

The remainder of the paper is organized as follows: Section 2 describes the translation system, Section 3 presents and discusses performance results and Section 4 discusses related work, followed by conclusions and future work in Section 5.

2 The OpenMP to Message-Passing Translator

The translator converts a C program using OpenMP parallel constructs into a message-passing program. This is accomplished by first changing the OpenMP program to a SPMD form; the resulting code represents a *node program* for each thread[3] that operates on partitioned data. The SPMD-style representation has the following properties: (i) the work of parallel regions is evenly divided among the processes; (ii) serial regions are redundantly executed by all participating processes; and (iii) the virtual address space of shared data is replicated on all processes. Shared data is not physically replicated - only the data actually accessed by a process is physically allocated on that process. Next, the compiler performs array data flow analyses, which collect written and read shared array references for the node program in the form of symbolic expressions. The compiler inserts function calls to pass these expressions to the runtime system. Finally, at each synchronization point, the runtime system uses the expressions from the compiler to compute the inter-thread overlap between written and read data and determines the appropriate MPI communication messages.

The translation system guarantees that shared data is coherent at OpenMP synchronization constructs only, as defined by OpenMP semantics. A data race that is not properly synchronized in the input OpenMP program might lead to unspecified results of the translated code.

The translation system is implemented using the Cetus compiler infrastructure. We also developed the runtime library. To support interprocedural analysis, we use subroutine inline expansion. Figure 1 shows an example of an input OpenMP code and the translated code.

2.1 SPMD Generation

The input OpenMP program is converted to an initial SPMD form that is suitable for execution on distributed-memory systems. First, the compiler parses OpenMP directives to identify serial and parallel regions. Second, it identifies shared variables in these regions. Third, it partitions the iteration space of each parallel loop among participating processes. The following subsections describe in detail these three steps.

2.1.1 OpenMP Directives Parsing: The compiler parses OpenMP directives in the input program and represents them internally as *Cetus annotations*. Cetus annotations identify code regions and express information such as shared and private variables as well as reduction idioms.

The compiler inserts a barrier annotation after every OpenMP work-sharing construct that is terminated by an implicit barrier, to indicate the presence of that barrier. No barriers are present at `nowait` clauses. We refer to the code executed between two consecutive barriers as a *SPMD block*. The `omp master` and `omp single` constructs in parallel regions are treated as serial regions.

[3] OpenMP thread corresponds to MPI process

```
                                        for (id = 0; id < nprocs; id++) {
                                          ompd_def(1, id, A, lb2[id], ub2[id]);
                                          ompd_use(1, id, A, lb1[id]-1, ub1[id]+1);
                                        }

for (step = 0; step < nsteps; step++) {  for (step = 0; step < nsteps; step++) {
  x = 0;                                   x = 0;

  /* loop 1 */                            /* loop 1 */
  #pragma omp parallel for reduction(+:x)  #pragma cetus parallel for shared(A) private(i)
  for (i = 0; i < M; i++)                reduction(+:x)
    x += A[i] + A[i-1] + A[i+1];           x_tmp = 0;
                                           for (i = lb1[proc_id]; i <= ub1[proc_id]; i++)
  printf("x = %f\n", x);                     x_tmp = A[i] + A[i-1] + A[i+1];
                                           #pragma cetus DEF()
  /* loop 2 */                            #pragma cetus USE(A[lb1[proc_id]-1:ub1[proc_id]+1])
  #pragma omp parallel for                 #pragma cetus barrier /* barrier 0 */
  for (i = 0; i < N; i++)                  ompd_allreduce(&x_tmp, &x, ...);
    A[i] = ...;
}                                          printf("x = %f\n", x);

                                           /* loop 2 */
                                           #pragma cetus parallel for shared(A) private(i)
                                           for (i = lb2[proc_id]; i <= ub2[proc_id]; i++)
                                             A[i] = ...;
                                           #pragma cetus DEF(A[lb2[proc_id]:ub2[proc_id]])
                                           #pragma cetus USE(A[lb1[proc_id]-1:ub1[proc_id]+1])
                                           #pragma cetus barrier /* barrier 1 */
                                           ompd_ptp(1);
                                         }
```

Fig. 1. Typical code of an OpenMP program and its translated code: The runtime library functions ompd_allreduce and ompd_ptp compute and perform the necessary communication, based on the array sections given by the ompd_def/use calls. These calls have been hoisted before the "step" loop, as they are loop invariant in this example. Cetus pragma annotations are comments only.

The compiler identifies reduction operations. Explicit `omp reduction` clauses in the source program are directly represented by a reduction annotation. A widely used practice in OpenMP programs is to code array or scalar reductions using the `omp critical` or `omp atomic` directive constructs, where each thread updates the global copy of the shared data using its own local copy. The compiler also identifies these reduction patterns and provides the corresponding critical section with a reduction annotation. The translator uses an existing algorithm in the Cetus infrastructure to recognize these patterns. The compiler then converts all these reductions to MPI reduction operations, as described in Section 2.3.1. In our benchmarks, the compiler recognized all `omp critical` sections as reduction patterns. Figure 1 shows how the OpenMP directives in the input code are represented using Cetus annotations in the translated code.

2.1.2 Shared Data Identification: In preparation for communication analysis, the compiler must identify all shared data. Variables in OpenMP parallel regions are shared by default - the OpenMP data clauses, such as `omp private` and `shared` are not sufficient to identify shared variables. The compiler uses an

interprocedural algorithm described previously [7] for finding shared variables in parallel regions. The compiler updates each Cetus annotation to store the shared variables and private variables to be used in subsequent translation steps.

2.1.3 Work Partitioning: The compiler uses block distribution to divide the iteration space of a parallel loop into chunks of approximately equal size, and then associates each chunk with a process. The compiler inserts the code to perform this partitioning as early as possible in the program. Under OpenMP semantics, using block distribution, even in the presence of an OpenMP `schedule` clause, always leads to correct results[4]. Block distribution also simplifies the symbolic expressions representing the summarization of affine array subscripts. Before partitioning a parallel loop, the compiler checks if all array subscripts for shared write accesses are affine expressions. In the presence of a non-affine write array subscript (e.g., an indirect array access), the parallel loop is not partitioned, i.e., conservatively serialized. In ongoing work, we are developing techniques for handling irregular write memory accesses.

2.2 Array Data Flow Analyses

Using Data Flow Analyses, the compiler collects information about written and read shared array references in the SPMD program that may need to be communicated. For each barrier N, the compiler symbolically summarizes shared written array references in the preceding SPMD block that reach barrier N and future, shared read array references exposed to barrier N. The compiler passes this information to the runtime system, which generates communication, as explained in Section 2.3.

The compiler first constructs a *parallel control flow graph (PCFG)*. The PCFG is essentially the control flow graph (CFG) generated for a node program. Every statement is represented by a node in the PCFG. Every Cetus annotation is attached to its corresponding node, except for barriers, which are converted into special nodes. The compiler then performs the following data flow analyses: *Reaching-All Definitions* analysis to compute *DEF* sections and *Live-Any* analysis to compute *USE* sections. A DEF section summarizes shared written array references produced in the SPMD block preceding barrier node N. Because serial regions are executed redundantly, their produced (written) shared data are available to all future consumer processes; therefore DEF sections need not include those references.

The array data flow analyses are based on Cetus array section analysis [6], which summarizes the set of array elements (i.e., the sub-arrays) that are written or read by a program statement. The array section analysis makes use of symbolic range analysis, which collects, at each statement, a map from integer-typed scalar variables to their symbolic value ranges, which are represented by a symbolic lower bound (lb) and upper bound (ub). DEF and USE sections are represented

[4] See Chapter 2.5.1 of the OpenMP Specification. The other scheduling methods will be supported in our ongoing work.

using Cetus section [lb:ub] symbolic expression. Cetus array sections conservatively describe non-affine array subscripts, such as indirect accesses. Live-Any analysis overestimates by assuming all data is read when summarizing a non-affine read array subscript using the infinity section expression [-INF:+INF]. Communication is optimized for [-INF:+INF] USE data sections by using MPI collective communication operations (see Section 2.3.1). Strided affine subscripts are represented using Cetus unit-stride array section and thus overestimated as well. Overestimation for USE sections can cause extra communication but no incorrect behavior. Note that DEF sections are never overestimated; as an `omp for` containing irregular write array subscripts is serialized (see section 2.1.3).

After performing the array data flow analyses, the entry and the exit of each node N in the PCFG have the following information: (i) $shared\_write\_set(N)$ is the set of shared written arrays in the parallel region prior to node N; (ii) $shared\_read\_set(N)$ is the set of shared arrays that is upward exposed at node N; (iii) tuples of DEF sections $\langle DEF_i \rangle_V, 0 \leq i \leq (p-1)$, DEF_i is the DEF section computed for process i and p is the total number of participating processes, $\forall V \in shared\_write\_set(N)$; (iv) tuples of USE sections $\langle USE_j \rangle_W, 0 \leq j \leq (p-1)$, USE_j is the USE section computed for process $j, \forall W \in shared\_read\_set(N)$. The compiler represents a tuple of array sections using one symbolic expression, where this expression is a function of the processor number (proc_id). The compiler inserts Cetus annotations to show computed DEF and USE sections' tuples at barriers in the output code as shown in Figure 1.

2.3 Communication Generation

At a barrier, inter-process communication is needed if the DEF section for one process overlaps with a USE section for any other process. The compiler has identified DEF and USE sections for all barriers. The compiler inserts function calls to pass these sections to the runtime system; it also inserts a function call at each barrier notifying the runtime system to generate communication. The runtime system's role is to compute the inter-process overlapping array sections and to generate MPI point-to-point communication if the overlap is non-empty. An exception of this is when generating MPI collective communication; the compiler recognizes that using collective communication instead of using point-to-point communication is possible. The following subsections discuss in detail the process of generating communication for each case.

2.3.1 Collective Communication Generation: The compiler generates collective communication in two cases: at reduction operations and at barriers that have a [-INF:+INF] USE section overlapping with a DEF section. These cases are recognized at compile time, where the compiler inserts function calls to generate collective communication at the corresponding synchronization points. The compiler does not need to inform the runtime about DEF and USE sections in separate function calls; this information is passed in the arguments of the collective communication function call.

The `ompd_allreduce()` call is used for reductions, which invokes `MPI_Allreduce()` collective communication. In the case of a reduction clause in an `omp for` loop, the compiler creates a local copy of the reduction variable, which is updated in the body of the `omp for` loop. An `ompd_allreduce()` is inserted after the `omp for` loop to combine the local copies into the global reduction variable. Figure 1 shows an example of this case. If a reduction originates from a critical section (see Section 2.1.1), the compiler replaces the critical section code with `ompd_allreduce()`.

The `ompd_allgatherv()` is used at barriers that have a DEF section overlapping with a [-INF:+INF] USE section, which calls `MPI_Allgatherv()` collective communication. `MPI_Allgatherv()` gathers the data chunks modified in each process and distributes them to all processes. Note that the relevant information about DEF sections is passed in the arguments of the `ompd_allgatherv()`, as they are used as arguments for the `MPI_Allgatherv()`.

2.3.2 Point-to-Point Communication Generation: For each barrier, the compiler passes DEF and USE symbolic sections to the runtime system. The runtime system computes the inter-process intersections of DEF and USE sections and generates the needed MPI point-to-point communication. At each barrier N, for each shared array V $\in$ (shared_write_set(N) $\cap$ shared_read_set(N)) and $\langle USE_j \rangle_V$ is not [-INF:+INF], the compiler inserts the following runtime function calls: (i) *ompd_def()* to pass DEF_i section computed for each process i; (ii) *ompd_use()* to pass USE_j section computed for each process j; (iii) *ompd_ptp()* to notify the runtime system that a barrier N has been reached. Each barrier is given a unique integer identifier by the compiler.

The translator exploits the *repetitiveness* property of DEF and USE sections to make the final selection of communication at runtime more efficient. A repetitive section is a section whose symbolic expression is invariant with respect to a outer serial loop(s), i.e., the same shared array references are accessed by the same process during the execution of all iterations of outer serial loops. An algorithm developed in prior work [8] is used at compile-time to verify the repetitiveness property of array sections. The compiler inserts function calls *ompd_def()* and *ompd_use()* prior to the outermost serial loop at which array sections are repetitive. This information needs to be passed only once; the runtime system can repetitively reuse these sections. In our tested benchmarks, the compiler has recognized all array sections as repetitive sections.

When reaching a barrier B, the function call *ompd_ptp()* triggers the runtime system to determine and generate MPI point-to-point communication. Algorithm 1 depicts the underlying algorithm of this function. In the algorithm, the intersections $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ represent array sections that need to be sent and received by a process. Because DEF and USE sections are repetitive, the intersections $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ are also repetitive. The runtime system utilizes this property to compute these intersections once and then repeatedly uses them for subsequent iterations of outer loops. The *is_updated* flag indicates if DEF or USE sections have changed. The *is_updated* flag is set to

Algorithm 1 Determining and generating point-to-point communication messages. The total number of processors is p.

```
if is_updated = TRUE then
  for j = 0 → p − 1 do
    if j ≠ proc_id then
      send_msg_j ← DEF_{proc_id} ∩ USE_j
      receiver_j ← j
    end if
  end for
  for i = 0 → p − 1 do
    if i ≠ proc_id then
      receive_msg_i ← USE_{proc_id} ∩ DEF_i
      sender_i ← i
    end if
  end for
  is_updated ← FALSE
end if
count ← 0
for all send_msg_j do
  if send_msg_j ≠ EMPTY then
    MPI_Isend(send_msg_j, receiver_j, ...)
    count ← count + 1
  end if
end for
for all receive_msg_i do
  if receive_msg_i ≠ EMPTY then
    MPI_Irecv(receive_msg_i, sender_i, ...)
    count ← count + 1
  end if
end for
MPI_Waitall(count, ...)
```

$TRUE$ by the *ompd_def()* and *ompd_use()* functions and set to $FALSE$ when the intersections are computed. These intersections can be reused as long as *is_updated* is $FALSE$, i.e., *ompd_def()* and *ompd_use()* are not re-invoked.

The runtime maintains $\langle send\_msg_j \rangle$ and $\langle receive\_msg_i \rangle$ sections information by storing a list of lower and upper bounds and their corresponding barrier identifier. The translator uses asynchronous point-to-point communication to overlap the processes' waiting times. Figure 1 shows the inserted function call to the runtime library at $Barrier1$. No function call is inserted at $Barrier0$ because shared_write_set($Barrier0$) ∩ shared_read_set($Barrier0$) is empty.

3 Performance Evaluation

This section evaluates our translation system on five benchmarks: The JACOBI and SPMUL kernel benchmarks and CG, EP and FT from the NAS Paral-

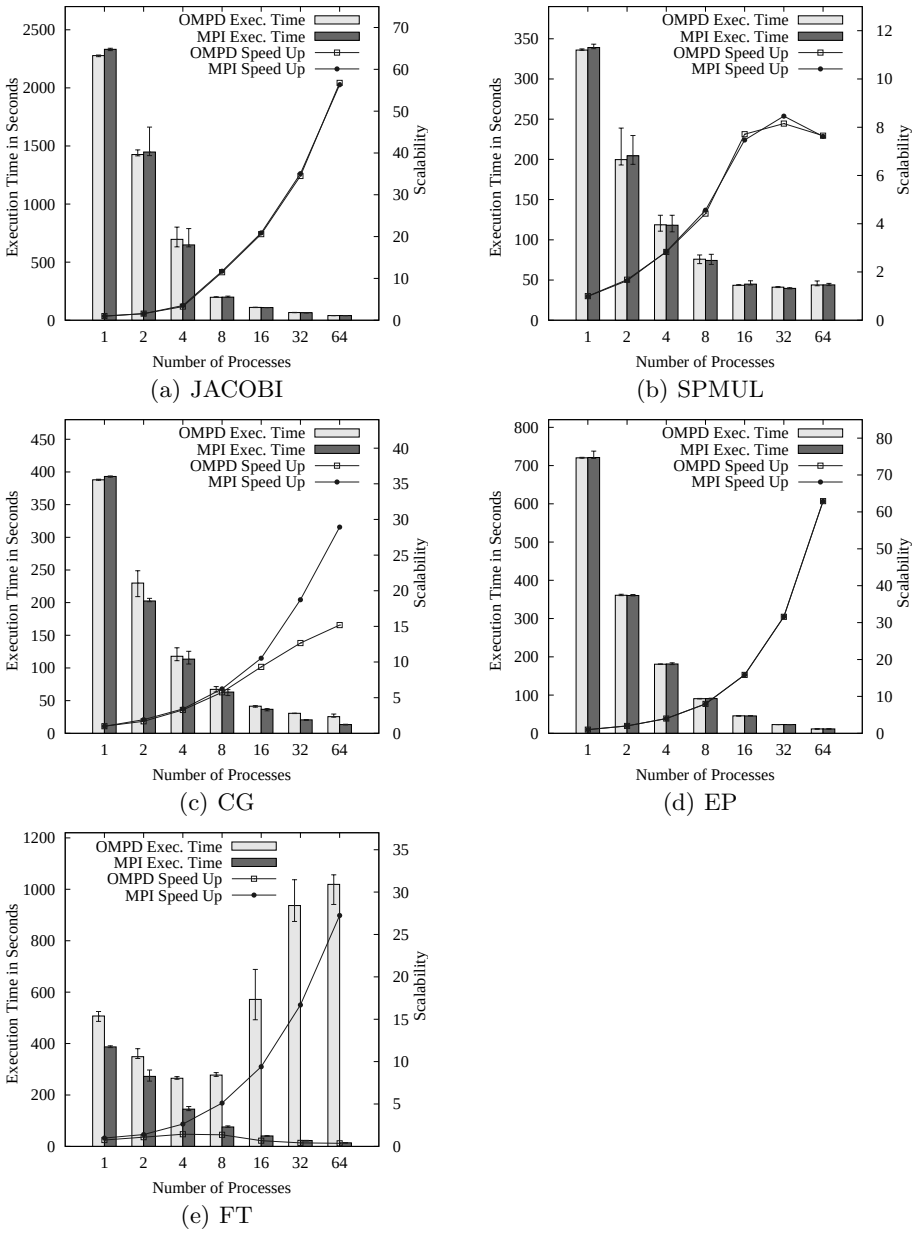

Fig. 2. Performance comparison between translated and MPI programs: the translated JACOBI, SPMUL and EP perform very close to the hand-tuned MPI counterparts. The performance difference in the CG programs results from different data partitioning schemes of the translator and MPI programs. The FT benchmark exposes opportunities for future work.

lel Benchmarks (NPB) 2.3 [9, 10]. Each translated benchmark is compared to its hand-tuned MPI counterpart. We use publicly available OpenMP and MPI benchmarks. For JACOBI, we used an OpenMP version [11] and a fast MPI version [12] that uses asynchronous communication. For SPMUL, the main computation part of the OpenMP version is based on an implementation using compressed sparse rows (CSR) in SPARSKIT [13], and the MPI version is also available online [14].

We performed subroutine inlining and found that it did not significantly affect our performance; the difference in execution time between the original and inlined programs is less than 0.1% for all benchmarks except for FT, where it is around 4%.

3.1 Experimental setup

We evaluated the performance on 32 nodes of a community cluster with two MPI processes per node, for a total of 64 MPI processes. The nodes are connected by an InfiniBand network, which provides 10 Gbps of bandwidth. Each node has two quad-core Intel Xeon E5410 processors running at 2.33Ghz with 6MB L2 caches per processor and 16GB of memory. The system is running a 64-bit Linux kernel, version 2.6.18, and MVAPICH2 version 1.5 MPI. We compiled all programs with gcc64 4.4.0 at optimization level 3.

We timed each benchmark ten times and recorded average, minimum and maximum execution times. Figure 2 shows the average execution time for each benchmark, with error bars indicating minimum and maximum times. We attribute the variations primarily to network traffic of other jobs running on different nodes of the system. Graphs have an axis labeled "scalability" – this is the ratio of the execution time of the serial version of the benchmark running on one processor to the execution time of the translated benchmark running on the number of processors indicated on the X axis.

3.2 JACOBI Kernel

JACOBI is a kernel benchmark that solves Laplace equations using Jacobi iteration. It uses two 2,048 $\times$ 2,048 matrices with 100,000 iterations. Figure 2(a) compares the performance of the translated program and the hand-tuned MPI variant. The runtime intersection between USE and DEF data results in the same communication volume that the MPI version generates. The MPI version is one of the fast versions that use asynchronous communication among neighbor processes. Since our runtime system also uses asynchronous communication for all messages, we are able to match the MPI variant's performance.

3.3 SPMUL Kernel

SPMUL is a kernel benchmark that performs sparse matrix-vector multiplications. We used a 100,000 $\times$ 100,000 sparse matrix as input. Figure 2(b) compares

the performance of the translated program and the hand-tuned MPI program. The former program has two barriers. One causes point-to-point communication to be generated and the other causes collective communication to be generated. The communication volume at the first barrier is zero because the intersection between DEF and USE data sections is empty; no communication is performed. At the second barrier, the communication is done by `ompd_allgatherv()` because the code has irregular data reads on arrays via an indirection vector, so the USE data section is treated conservatively as [-INF:+INF]. The MPI program uses block-striped partitioning and has one communication point where a collective communication using `MPI_Allgatherv()` occurs. Even though the translated program has two barriers while the MPI program has one, the actual runtime communication patterns of the two programs are identical; the communication volume at the first barrier is zero.

3.4 CG Benchmark

The CG benchmark implements a conjugate gradient method, with a sparse matrix-vector multiplication taking most of the time. The benchmark has irregular reads on arrays via an indirect vector because of its unstructured grid computations. In addition to the irregular reads, there is an `omp for` loop that has irregular writes. Our translator treats this loop as a serial region, as described in Section 2.1.3. This serial loop execution does not significantly affect the benchmark performance because it exists outside of the outermost time-stepping serial loop.

The OpenMP version of the CG benchmark includes several reduction clauses in `omp for` work sharing constructs. The reductions are translated into `ompd_allreduce()` communication by the translator.

The translated and the MPI programs have different data partitioning schemes. The MPI version partitions the sparse input matrix using 2-D block distribution, whereas our translation scheme partitions the input matrix using a simple 1-D block distribution. This is because only the outer-most loop of the matrix-vector multiply computation is parallelized in the OpenMP version. Compared to the 1-D distribution, the 2-D block distribution of the input matrix requires a smaller number of processors for reduction communication; it comes at the expense of one additional transpose operation of the input matrix at the end of the matrix-vector multiplication. The overhead of the 2-D distribution's transpose operation is not proportional to the number of processors. Therefore, the 1-D distribution is superior for small numbers of processors. As the number of processors increases, however, the 2-D distribution shows better communication performance.

To further improve the performance of the translated version, optimization using 2-D distribution of the input matrix is needed, and it requires the recognition of matrix-vector multiplication patterns in the source program.

3.5 EP Benchmark

The EP benchmark is a highly parallel kernel, often used to explore the upper limit of floating point performance of a parallel system. The OpenMP version has one `omp critical` section that performs an array reduction and our implementation transforms it into an `ompd_allreduce()` function call. In addition, one `omp for` work sharing construct has a reduction clause on scalar variables; these reductions are translated into two `ompd_allreduce()` functions. The translated code is similar to the MPI version of the EP benchmark. The MPI version uses three `MPI_Allreduce()` function calls. The resulting performance of the translated code is nearly identical to the hand-tuned MPI variant, as shown in Figure 2(d).

3.6 FT Benchmark

The FT benchmark is a kernel that solves a 3-D partial differential equation using Fast Fourier Transforms. The code tests the communication performance of a system because it is very message intensive. As shown in Figure 2(e), the MPI version outperforms the translated variant substantially. Our version shows speedup on up to 8 processors, after which the communication overhead dominates.

The input FT code for our translator has a manual modification splitting `dcomplex` into `real` and `imaginary` variables. This is done because the current array section analysis does not support structured data types. The split affects cache behavior and generates two separate MPI communication calls. We measured the split version and the structured version of the translated FT program and found that the split version is around 20% slower than the structured version on one MPI process.

We identified two issues related to communication. First, the MPI version uses collective communication, but our implementation uses less efficient point-to-point communication. Thus, making the runtime detection of the collective communication pattern is one of the next goals of our system development. Second, the translated code uses two additional communications not present in the MPI version. Closer analysis revealed that performance could be improved through partitioning methods on different iteration spaces of nested parallel loops that consider data affinity between processes. Furthermore, more accurate analysis of indirect array accesses – including the use of runtime methods – could improve on the conservative handling of array sections.

We identified compiler transformation and runtime techniques based on the knowledge above. When we applied them manually, the translated version achieved almost identical performance of the MPI version. The automation of those techniques is beyond the scope of this paper.

4 Related Work

An approach to extending the ease of shared memory programming to distributed memory platforms, such as clusters, is the use of a Software Distributed

Shared Memory (SDSM) system. SDSM is a runtime system that provides a shared address space abstraction on distributed memory architectures. Researchers have proposed numerous optimization techniques to reduce remote memory access latency on SDSM. Many of these optimization techniques aim to perform pro-active data movement by analyzing data access patterns either at compile-time or at runtime. In compile-time methods, a compiler performs reference analysis on source programs and generates information in the form of directives or prefetch instructions that invoke pro-active data movement at runtime [15]. The challenges for compile-time data reference analysis are the lack of runtime information (such as the program input data) or complex access patterns (such as non-affine expressions). By contrast, runtime-only methods predict remote memory accesses to prefetch data based on recent memory access behavior [16]. These methods learn the communication pattern in *all* program sections and thus incur overheads even in those sections that a compiler could recognize as not being beneficial. The idea of combined compile-time/runtime solutions has been studied [8, 17–19]. However, all these page-based SDSM approaches incur inherent overheads when detecting shared data accesses using a page-fault mechanism. Another drawback of this model is substantial false-sharing due to the page granularity at which SDSMs operate. Our system does not rely on any expensive operating system support, such as a page-fault mechanism; it generates MPI messages to communicate any size of array sections, which avoids false-sharing.

Another important contribution towards a simpler programming model for distributed memory machines was the development of High Performance Fortran (HPF) [1]. There are significant differences between our OpenMP-to-MPI translation approach and that of HPF. Even though HPF, like OpenMP, provides directives to specify parallel loops, HPF's focus is on the data distribution directives and automatic parallelization guided by those directives. Data partitioning is explicitly given by these directives and computation partitioning is guided by them [20]. Data typically has a single owner, and computation is performed by the owner of written data, i.e., the *owner computes* rule. In contrast to HPF, OpenMP has no user-defined data distribution input. Thus, computation partitioning is not derived from data distribution information. Instead, computation is distributed among processors based on OpenMP directives. Therefore, unlike most HPF implementations, our execution model has no concept of the *owner computes* rule.

PGAS (Partitioned Global Address Space) languages, such as UPC [3], Co-array Fortran [4], Titanium [21], and X10 [22], 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 programmer needs to specify the affinity between threads and data. In our work, the programmer writes a standard OpenMP shared memory program and our system translates this program into message-passing form for distributed memory platforms.

5 Conclusions and Future Work

The vision of a programming system that offers a shared address space to the user and executes applications efficiently on a distributed architecture has been elusive so far. High-Performance Fortran and Software Distributed-Shared-Memory systems have never received wide-spread acceptance. UPC and Co-array Fortran are current systems that involve the user in decisions about data placement on the distributed architecture. By contrast, the presented approach shows that unmodified OpenMP applications can be translated to execute on a cluster platform at performance levels close to hand-coded MPI. Our system is the first automatic translator, and supporting runtime system, that proves the value of automatic translation of shared memory OpenMP programs to execute on distributed memory machines. The results indicate that a high-productivity and high-performance programming interface for modern distributed machines is possible. As with all work on compilers and tools to support high performance parallel programming, additional research will yield better results on a wider range of programs.

In ongoing research, we aim to increase the scope of applications that the system can handle. As well, immediate benefits can come from improved recognition of collective operations at both compile time and during the program execution, and from exploiting data affinity and advanced work partitioning schemes. An advanced hybrid compiler/runtime technique for improving the accuracy of array sections is also under development.

References

1. High Performance Fortran Forum: High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex. (1993)
2. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer **29**(2) (February 1996) 18–28
3. UPC Consortium: UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Laboratory (2005)
4. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum **17**(2) (1998) 1–31
5. Basumallik, A., Eigenmann, R.: Towards Automatic Translation of OpenMP to MPI. In: ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing, New York, NY, USA, ACM Press (2005) 189–198
6. Bae, H., Bachega, L., Dave, C., Lee, S., Lee, S., Min, S., Eigenmann, R., Midkiff, S.: Cetus: A Source-to-Source Compiler Infrastructure for Multicores. In: Proc. of the 14th International Workshop on Compilers for Parallel Computing (CPC'09). (January 2009)
7. Min, S., Basumallik, A., Eigenmann, R.: Optimizing OpenMP programs on Software Distributed Shared Memory Systems. International Journal of Parallel Programming **31**(3) (2003) 225–249
8. Min, S., Eigenmann, R.: Combined Compile-time and Runtime-driven, Pro-active Data Movement in Software DSM Systems. In: LCR '04: Proceedings of the 7th

Workshop on Languages, Compilers, and Run-time support for scalable systems, New York, NY, USA, ACM Press (2004) 1–6
9. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. (1991)
10. Satoh, S.: NAS Parallel Benchmarks 2.3 OpenMP C version [Online]. Available: `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp` (2000)
11. Andrews, G.R.: Foundations of Parallel and Distributed Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
12. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: Point-to-Point Communication. In: MPI: The Complete Reference (Vol. 1). MIT Press (1998) 56–65
13. Saad, Y.: SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455 (June 1994) Version 2.
14. Bhardwaj, D.: Description for Implementation of MPI Programs [Online]. Available: `http://www.cse.iitd.ernet.in/~dheerajb/MPI/Document/tp.html`
15. Dwarkadas, S., Cox, A.L., Zwaenepoel, W.: An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In: Proc. of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII). (1996) 186–197
16. Bianchini, R., Pinto, R., Amorim, C.L.: Data prefetching for software DSMs. In: the 12th International Conference on Supercomputing. (1998) 385–392
17. Viswanathan, G., Larus, J.R.: Compiler-directed Shared-memory Communication for Iterative Parallel Applications. In: Supercomputing. (Nov. 1996)
18. Keleher, P., Tseng, C.W.: Enhancing Software DSMs for Compiler-Parallelized Applications. In: Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97). (1997)
19. Keleher, P.: Update Protocols and Iterative Scientific Applications. In: Proceedings of the First Merged Symposium IPPS/SPDP (IPDPS'98). (1998)
20. Gupta, M., Midkiff, S., Schonberg, E., Seshadri, V., Shields, D., Wang, K.Y., Ching, W.M., Ngo, T.: An HPF compiler for the IBM SP2. In: Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), New York, NY, USA, ACM (1995) 71
21. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. In: In ACM. (1998) 10–11
22. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications. (OOPSLA '05), New York, NY, USA, ACM (2005) 519–538