Is OpenMP for Grids ?

Rudolf Eigenmann[†], Jay Hoeflinger[§], Robert H. Kuhn[§], David Padua[‡],
Ayon Basumallik[†], Seung-Jai Min[†], Jiajing Zhu[‡]

[†]Purdue University
[§]KAI Software/Intel Americas, Inc.
[‡]University of Illinois at Urbana-Champaign

Abstract

This paper presents an overview of an ongoing NSF-sponsored project for the study of runtime systems and compilers to support the development of efficient OpenMP parallel programs for distributed memory systems. The first part of the paper discusses a prototype compiler, now under development, that will accept OpenMP and will target TreadMarks, a Software Distributed Shared Memory System (SDSM), and Message-Passing Interface (MPI) library routines. A second part of the paper presents ideas for OpenMP extensions that enable the programmer to override the compiler whenever automatic methods fail to generate high-quality code.

1. Introduction

One of the main objectives of parallel software research has been the development of a standard programming methodology for the development of efficient programs for all classes of parallel machines. Such standardization would reduce the effort needed to train programmers, facilitate the porting of programs, and, in general, would reduce the burden of adopting parallel computing.

An important contribution in this direction was the development of High Performance Fortran (HPF) [4] which extended Fortran 90 with directives to specify parallel execution, data distribution, and data alignment. Although practically all research on HPF compilers assumed that the target was distributed memory machines, it is possible, and probably much easier, to target HPF to shared memory systems. Unfortunately, HPF has not received widespread acceptance and parallel programming seems to have converged to two approaches that differ significantly from each other.

By far the most popular way of programming parallel machines today, especially clusters and distributed memory machines in general, is to write SPMD (Single Program Multiple Data) programs and use Message-Passing Interface (MPI) library routines [6] for communication and synchronization. The second approach, which dominates when the target machine is a Symmetric Multiprocessor (SMP) with a few processors, is to use thread libraries or OpenMP [3] to write parallel programs assuming a shared memory model. OpenMP resembles HPF because of its reliance on directives. However, the OpenMP standard differs from HPF in that it deals almost exclusively with control flow and synchronization and has practically no mechanism to control data placement and alignment.

Of the two approaches, the former is seen as a low level programming model to the point that MPI has been called the assembly language of parallel programming. Clearly, message-passing programming has the advantage that it gives the programmer direct and explicit control of the communication between threads. However, the complexity of subscripts that arise when arrays are distributed manually and the difficulty of changing distributions and, in general, modifying message-passing parallel program makes the message-passing programming model inconvenient and costly.

Given this state of affairs, a natural course of action is to try to unify the two approaches into a universal parallel programming paradigm that can be used for all classes of parallel systems available today from small SMPs to wide-area distributed computer grids. OpenMP, with the appropriate extensions, seems a natural candidate for such a unifying role. Both OpenMP and thread libraries bring the programming advantages of the shared memory model, but OpenMP has the additional advantage of enforcing a nested structure in the parallel program. This last consideration gives OpenMP an advantage over thread libraries.

Although OpenMP was developed to program shared memory machines, we believe it is possible to use OpenMP to generate efficient programs for distributed memory clusters and computer grids. Clearly, to achieve this goal the appropriate runtime systems, OpenMP extensions, and compiler techniques must be developed.

Although the replacement of MPI with OpenMP will probably happen first for conventional distributed memory machines, we believe that grids are also an important target for this project. In fact, grids have matured from just a collaborative environment to a resource where users are actually thinking of running large parallel applications. The state-of-the-art tools available to parallel programmers to harness this resource appears to be MPI variants such as MPICH-G. However, programming at a higher abstraction level is perhaps even more important in the grid scenario due to the complexity involved in the programming of unrelated systems.

A possible approach to implement OpenMP is to use a Software Distributed Shared Memory (SDSM) system such a TreadMarks [1] to create a shared memory view on top of the target system. By following this approach the implementation of OpenMP on distributed memory systems becomes equivalent in difficulty to implementing OpenMP on an SMP machine. The drawback is that the overhead typical of SDSMs can affect speedup significantly.

A way to reduce the overhead is to translate OpenMP programs so that the SDSM system is used only to handle the communications due to arrays with access patterns that are irregular or unknown to the compiler, while communications due to regular access patterns is handled by message-passing libraries such as MPI. This can be achieved by applying compiler techniques similar to those developed by HPF. This approach does not suffer from the same overhead problems as the SDSM approach in the case of regular memory access patterns and can be easily combined with the SDSM approach to handle irregular accesses.

The development of an effective compiler methodology to map OpenMP codes onto a runtime environment containing message passing routines, a SDSM system, and other forms of runtime support could be a major challenge. One reason is that not much is known about the effectiveness of today's compiler technology in generating message-passing code from shared memory programs. Much testing and evaluation of existing compiler techniques for message generation is still needed and probably these techniques will have to be extended to achieve the desired result. Also, the combination of several approaches introduces important questions that need to be studied. For example, besides using SDSM support it is also possible to use inspector-executor techniques to handle irregular access. It is not clear how these two ways to deal with irregular accesses compare in terms of implementation cost, efficiency, and portability.

Any effective compiler system should allow users direct control of the code generated, especially when the compiler technology is not mature. To this end, we propose to extend OpenMP to allow users direct control of message-passing code generation, computation distribution, and data alignment. This is important because the compiler, especially the earlier versions of it, is not expected to adequately handle all conceivable situations. Providing direct control through extensions to OpenMP will make it possible for the programmers to take advantage of the compiler in order to avoid the complexities of message-passing programming in a significant portion of the program. And in those, hopefully small, sections of the code where the compiler fails, the programmer will be able to intervene and produce efficient code.

Under NSF support we are developing a compiler that will accept an extended form of OpenMP and will generate parallel code that will take advantage of the most appropriate runtime support available. So far, we have only considered software runtime support, but we plan to look at hardware support in the future. As discussed below, it is reasonable to expect that successful software mechanisms will evolve (at least partially) into hardware form.

This paper describes our OpenMP compiler. The first version of this compiler will accept standard OpenMP programs. We have already developed practically all components of this first version of the compiler and only their integration in a complete system remains. A second version of the compiler will include a number of additional optimizations along the lines of those discussed in Section 2.6, and will accept OpenMP extensions as discussed in Section 3.

Although this paper does not give a complete answer to the question posed in the title, it presents a clear plan to find such an answer. We believe that, in order to be able to answer YES, a complete program development and execution system has to be developed, and it has to be shown that it enables the development of efficient parallel code for distributed memory systems. The rest of this paper presents some components of such a program development/execution system. Section 2 discusses the hybrid communication model and compiler techniques to take advantage of this model, which will be incorporated in the first version of our compiler. Additional compiler optimizations are discussed in Section 2.6, and Section 3 presents ideas about OpenMP extensions.

2 The OpenMP Compiler

2.1 The Hybrid Communication Model

Based on the Polaris system [2], we are developing a compiler that accepts conventional OpenMP and generates a program that follows a hybrid communication model. It uses a synthesis of the private memory assumed by MPI

programs today and shared memory models. The new model inherits the strong points from both existing models. By targeting this model, we believe that the compiler can generate efficient parallel code easily and automatically, on regular, irregular and mixed access pattern programs. We assume an environment that supports both message passing and a (hardware or software) shared memory mechanism.

In our hybrid communication model, we include both the private memory and the shared memory model. Since the most efficient mechanism for communicating data differs, depending on how the data is used within the code, we apply two different communication mechanisms to handle data with different communication features. The user data is classified by its usage patterns. We divide data into three classes: **private data**, **distributed data**, and **shared data**.

Private data consists of variables that are only accessed by a single thread. No communication is needed for this data. Some of the private data will be that identified in OpenMP *private* clauses and other will be identified by our OpenMP compiler as an optimization.

Distributed data consists of variables, with simple usage patterns, which might need to be communicated between threads. Communication of these variables is handled by the message passing mechanism. Since they have a simple usage pattern, the precise message passing code can be easily generated by the compiler automatically.

Shared data consists of variables with statically unknown or irregular access patterns. The consistency of shared variables across threads is handled by a shared memory mechanism. Shared data is data for which precise message passing code cannot be generated, due to lack of precise access information.

In this hybrid model, explicit message passing and the shared memory system are independent and equal-level mechanisms. They work separately on different data classes. The current target of our compiler is a system where shared memory is implemented in software via a SDSM system such as TreadMarks. Distributed data is located outside the shared memory space of the SDSM system, eliminating the overhead that would be otherwise necessary to manage it within the SDSM system.

2.2 Phases of the OpenMP Compiler

The framework of the prototype compiler algorithm to implement the hybrid model consists of three phases. We give a brief outline of the algorithm framework here, followed by a more detailed description in the next subsections.

Phase 1. Data classification and distribution: In this phase, the compiler classifies unprivatizable user data as either *distributed data* or *shared data*. Distributed data is divided among the threads, while shared data is apportioned according to the rules for the SDSM being used (TreadMarks replicates its shared data on each thread). The communication of distributed data will be handled by the explicit message passing mechanism, while the shared data will be handled via the SDSM mechanism.

The compiler analyzes data access patterns. Data having *simple-enough* access patterns is considered as distributed data. All other data in the program is considered as shared data, except for privatizable data identified in the private clauses of the OpenMP source program.

Phase 2. Computation Division: In this phase, the compiler divides the iteration space and assigns iterations to each thread. The code is organized in the SPMD model. This phase includes two subphases: computation division for distributed data and computation division for shared data. The iterations of parallel loops that contain distributed data are divided among the threads, based on, although not restricted to, the *owner-computes* rule. For the parallel loops that contain only shared data, the compiler assigns iterations to threads evenly.

Phase 3. Communication generation: Function calls for communications are generated and inserted in this phase. Like Phase 2, this phase consists of 2 subphases: for message passing communication and SDSM communication. The explicit message passing calls are generated at first for communicating the distributed data, and then SDSM calls for synchronizing the shared data are generated.

Notice that the framework is particularly designed for the hybrid communication model. It does not have an independent path for either message passing or SDSM code. The generation of message passing and SDSM primitives is performed jointly in a single pass.

2.3 Data classification and distribution

The source OpenMP program identifies private data and parallel loops. In Phase 1, we classify the remaining user data, which is involved in parallel loops, as distributed and shared data.

The algorithms used in our framework are quite simple, especially when comparing them to the algorithms that researchers have developed to automatically distribute and access data in distributed systems. This is one of the important advantages of our system. By applying an appropriate communication mechanism to each type of data (regular or irregular), we hope to generate efficient code, even by simple algorithms.

The algorithm is designed to operate in an incremental way. In the basic algorithm, only the data with very simple access patterns will be considered as distributed data, but the algorithm can be made more sophisticated and open to further optimization. New techniques can be added incre-

mentally to push more data into the distributed data set, to achieve better performance.

The algorithm analyzes the access pattern for every dimension of each reference with respect to the loop index. If the section of the array accessed by each iteration can be described at compile time using a simple notation, such as triplets, we say that the access is *regular*. Otherwise, we say that the access pattern is *irregular*. Then each dimension is classified as distributable or non-distributable depending on the type of loop that controls access to the dimension. Also, the type of distribution is determined by the iteration space of the loops controlling access to the dimension. More details about this algorithm can be found in [12].

2.4 Computation division

We discuss first how to translate parallel loops that only contain shared and private data. The common translation method for loop-parallel languages employs a microtasking scheme. In this scheme, execution of the program starts with the *master thread* which during initialization creates *helper threads* that *sleep* until they are needed. When a parallel construct is encountered, the master wakes up the helpers and informs them about the parallel code to be executed and the environment to be setup for this execution. Such microtasking schemes typically are used in shared memory environments, where communication latencies are low and where fully shared address spaces are available.

In contrast to shared memory environments, distributed architectures exhibit significantly higher communication latencies and they do not support fully-shared address spaces. In the TreadMarks SDSM system used in our work, shared memory sections can be allocated on-request. However, all thread-local address spaces are private – not visible to other threads. These properties question the benefit of a microtasking scheme because (1) the helper wakeup call performed by the master thread would take significantly longer and (2) the data environment communicated to the helpers could only include addresses from the explicitly allocated shared address space. Therefore, in our work we have chosen an SPMD execution scheme, as is more common in applications for distributed memory systems. The SPMD execution scheme is also necessary for program sections that access distributed data. In an SPMD scheme all threads begin execution of the program in parallel. Sections that need to be executed by one thread only must be marked as such explicitly and executed conditionally on the thread identification. In general, sequential program sections are executed redundantly by all threads, whereas in parallel regions the threads share the work.

For parallel loops that only contain references to shared data, we divide the iteration space evenly for each thread. Our SPMD translation of OpenMP parallel regions splits the worksharing constructs, modifying lower and upper bounds of the loops according to the iteration space assigned for each participating thread. Currently, we support static scheduling only. All parallel constructs containing at least one access to a shared variable are placed between a pair of barrier synchronizations. At these barriers, a coherent state of the shared memory is also maintained. During the parallel regions, the SDSM system supports release-consistency. The dual function of the barrier as synchronization and maintaining coherence is important to note. To maintain coherence, threads might need to execute expensive system calls, which increase in number with the amount of shared data written in the parallel region.

Parallel loops that contain distributed data or both distributed data and shared data are scheduled according to the layout of the distributed data, decided in Phase 1, and iterations are assigned to the thread that owns most of the left-hand side elements. Currently, each iteration is executed as a whole by a single thread, but this may be changed in the future.

In the cases where there are different access patterns in the same loop, we use an evaluation strategy to pick up the most frequently visited pattern as the owner pattern of the loop.

The translation of serial program sections is non-trivial. Since, in the SPMD scheme that we have adopted, all serial program sections will be executed by all threads, we need to identify serial program regions where this is not appropriate. If variables updated in the sequential region are shared, then they should not be redundantly updated by all threads. Instead, they are marked to be executed by the master only. Furthermore, after such a master-only section, a barrier synchronization is inserted. This ensures that other threads that subsequently read from these shared variables are properly delayed until the new values are correctly seen.

2.5 Communication Generation

In this phase, function calls for message-passing operations are generated and inserted. We compute the region of distributed data that needs to be transferred. Message passing calls are inserted to perform the communication for distributed data, in a copy-in/copy-out manner. Data that is read within a parallel loop nest is fetched from its owner before the loop. Data that is written within the loop must be written back to its owner afterward.

Figure 1 presents the algorithm for message passing call generation. For each reference x to array A in the source program, the compiler must determine a **send/receive** pair for communicating the data accessed but not owned by the involved threads. This is done by intersecting two internal representations, one describing the memory accesses made by x on thread P_i and the other describing the section of A

```
for(each reference x of distributed array A)
  access(P_j) = computeLMAD(x)
  for(each i and j, i≠j)
    if(x is read)
      mode = superset
    else
      mode = precise
    overlap =
      intersect(distribute(P_i), access(P_j), mode)
    if(overlap ≠ NULL)
      success =
        generate_message(overlap, i, j)
    if(!success) push_back(A)
  endfor
endfor
```

Figure 1. Message-Passing Communication

```
   real A(100)
!$omp distribute A(CYCLIC)
!$omp parallel do schedule(static)
   do i = 1, 100, 3
   ...  A(i) ...
   enddo
```

Figure 2. Loop accessing array A

owned by thread P_j. For our internal representation we use the LMAD (Linear Memory Access Descriptor) [7] notation which is more general than the conventional triplet notation. LMADs are usually able to capture precisely the accesses to a multi-dimensional array made by a sequence of multiply-nested loops, even when the subscript expressions are not affine.

To compute the intersection between the region of an array accessed in one thread and the region owned by another thread, we use the LMAD intersection algorithm which is described in [5]. This intersection is stored into the variable `overlap`. For example, consider an array, `A`, that has a `CYCLIC` distribution across four threads and is accessed in the loop in Figure 2. The region stored in thread P0 can be represented by the triplet `A(1:100:4)` and that stored in thread P2 by the triplet `A(3:100:4)`. The region accessed in thread P0 can be represented by the triplet `A(1:25:3)`. The intersection between the region accessed thread P0 and the region stored in thread P2 is `A(7:25:12)`

The intersect function requires a "mode" parameter that indicates what to do if the function cannot calculate the precise intersection. If the access is a read reference, it would not affect the correctness of the code if we fetch more data than needed from the other thread, so "superset" is specified for "mode". If the access is a write reference, the precise write-back is required. In that case, "precise" is specified.

region $\mathtt{L}_{ij}$: $\mathtt{A}(\mathtt{offset} : \mathtt{offset} + \mathtt{span} : 1)$
  if(my_id == i)
    send(A(offset), span+1, data_type, j)
  if(my_id == j)
    receive(A(offset), span+1, data_type, i)

(a) Transformation 1

region $\mathtt{L}_{ij}$: $\mathtt{A}(\mathtt{offset} : \mathtt{offset} + \mathtt{span} : \mathtt{stride})$
  MPI_type_vector(span/stride+1, 1, stride, data_type, NewT)
  if(my_id == i)
    send(A(offset), 1, NewT, j)
  if(my_id == j)
    receive(A(offset), 1, NewT, i)

(b) Transformation 2

Figure 3. Using LMADs to generate message passing code. Only simple LMADs, equivalent to triplets, are used here

Then, if the intersection operation cannot be done precisely, the intersection function will report a failure. For example, consider a complicated reference like `A(B(i), j-1)`, distributed as `A(*, BLOCK)`. If the access is a read, then we can conservatively fetch the whole column of `A`, so, the "superset" mode can be used. But if the access is a write, then the write-back must be precise. If the intersection procedure fails, due to being in "precise" mode, the array must be handled by the SDSM system and therefore the algorithm calls the `push_back` function to turn the array back to the shared data class.

A non-empty overlap result from the intersection operation indicates that communication is necessary. In the routine `generate_message`, we try to convert the overlap region to the proper message passing calls by transformations similar to those in Figure 3. The regions in Figure 3 are the non-empty results from intersecting the distribution regions on thread $\mathtt{P}_i$ with the regions accessed by thread $\mathtt{P}_j$.

In Figure 3, $\mathtt{L}_{ij}$ represents the intersection between the data stored in thread P_i and data read by thread P_j. The segment of code below the lines describing the $\mathtt{L}_{ij}$ regions are to be inserted before the loop reading the array so that it is executed by thread *i* and *j*. Figure 3 gives two kinds of transformations from LMAD (represented as triplets here for simplicity) to the equivalent MPI message passing calls. Figure 3 (a) shows the transformation from a dense region, which has the stride of 1, to the equivalent MPI calls. The thread where the data is stored sends the overlap data to the

thread that is going to access it. The beginning address of the message buffer is the offset of the overlap region. The length of the message buffer is `span+1`.

For the regions with $stride > 1$, as shown in Figure 3 (b), we build an MPI user data type to transfer the data. The MPI type vector can be used to build the user data type with the stride. It has the following interface [6]:

$MPI\_type\_vector($
$count, blocklength, stride, oldtype, newtype)$

The transformation in Figure 3 (b) can be extended to multidimensional regions.

For read references, the message passing calls are generated before the loop, to get the data before it is needed. For write references, the message passing calls are generated after the loop, to write the data back to its owner. The message passing codes in Figure 3 are for read references. The communication codes for write references can be generated by simply switching the sender and the receiver.

If some LMADs are too complicated to be converted into proper message passing calls, the `generate_message` function will return a failure status. If a failure happens, the `push_back` function will push the array back to the shared data class, letting the SDSM mechanism handle its communication.

2.6 Additional Optimization to Improve OpenMP Performance on SDSM Systems

The methods presented in this paper mark the starting point of a project that seeks answers to the following questions:

- Can OpenMP applications be executed efficiently on distributed systems and computer grids, and under what parameters?
- What are the compiler optimizations necessary to improve the efficiency of OpenMP applications running on distributed systems and computer grids?

In this section we discuss further the second question. We describe transformations that can optimize the performance of OpenMP programs on a SDSM system. Many of these techniques have been discussed in other contexts. It is their combined implementation in an OpenMP/SDSM compiler that we expect to have a significant impact. The realization of such a compiler is one objective of our ongoing project.

Data Prefetch: Prefetch is a fundamental technique for overcoming memory access latencies. Ideally, a data item x produced by thread A and consumed by B is sent to B as soon as it is produced by A. Prefetch actions are initiated by the receiver (B), in that the compiler moves the load of x upward in the instruction stream. Closely-related to prefetch is *data forwarding*, which is producer-initiated. Forwarding has the advantage that it is a one-way communication (A sends x to B) whereas prefetching is a two-way communication (B requests from A that x be sent). An important issue in prefetch/forwarding is to determine the optimal prefetch point. Choosing a prefetch point later than the earliest possible can be advantageous, as it reduces the need for prefetched data to be cached by the recipient and it enables the combination of several prefetch operations into a block transfer. Prefetch techniques have been studied previously albeit not in the context of OpenMP applications for SDSM systems. We expect that prefetch will significantly lower the cost of OpenMP END PARALLEL region constructs, as it reduces the need for coherence actions at that point.

Barrier elimination: Two types of barrier eliminations will become important. It is well known that the usual barrier that separates two consecutive parallel loops can be eliminated if permitted by data dependences [8]. Similarly, within parallel regions containing consecutive parallel loops, it may be possible to eliminate the barrier separating the individual loops. As barrier costs are high in SDSM systems, this optimization can improve program performance significantly.

In the OpenMP-to-SDSM transformations described in Section 2.4, barriers are also introduced in serial program sections, after write operations to shared data. The barrier can be eliminated in some situations, such as, if the shared data written are not read subsequently within the same serial section.

Data privatization: The advantage of private data and distributed data in a SDSM system is evident as they are not subject to costly coherence actions. An extreme of a SDSM program can be viewed as a program that has only private and distributed, and no shared, data. The necessary data exchanges between threads are performed by message operations (which can be implemented as copy operations in SMPs and NUMA machines). Many points are possible in-between this extreme and a program that has all OpenMP shared data placed in shared SDSM address space. For example, shared data with read-only accesses in certain program sections can be made "private with copy-in" during these sections. Similarly, shared data that are exclusively accessed by the same thread can be privatized during such a program phase.

We conducted an experiment where we manually applied privatization to read-only data in the *equake* benchmark. The program reads data in a serial region from a file. In the original code, since all the threads use the data, a shared attribute is given. However, we have modified the program so that the input data, which are read-only after the initialization, become private to all threads. The results show that even this simple optimization reduces the execution times substantially.

Page placement: Some SDSM systems place memory pages on fixed home threads, pages may migrate between threads, or pages are replicated (this is the case in TreadMarks). Fixed page placement leads to high overhead if the chosen home is not the thread with the most frequent accesses to this page. Migrating pages can incur high overhead if the pages end up changing their home frequently. Replicated pages can lead to high coherence traffic for keeping all page contents current. In all cases, the compiler can help direct the page management mechanism. It can estimate the number of accesses made to a page by all threads and choose the best scheme. By considering the type of accesses it can decide in which program phases page replication is beneficial and when single ownership is appropriate. Furthermore, the compiler can choose a page size that accommodates the desired data structures. We expect that the compiler's ability to consider both large and future program phases will give it a significant advantage over the SDSM page placement mechanism, which considers the program's recent past only. As a result, such compiler mechanisms will take over substantial control from the SDSM system.

Automatic data distribution: The method described in Section 2 is still quite primitive. Its effectiveness needs to be evaluated and the method extended, probably using some of the many methods discussed in the literature.

Data and loop reorganization: Data and loop transformation techniques for locality enhancement have also been researched extensively. We will study many of the proposed techniques. We expect that the transformation leading to data affinity will be among the most effective. That is, programs in which successive loops cause individual threads to access the same data will be most successful.

Adaptive optimization: A big impediment for all compiler optimizations is the fact that input data is not known at compile-time. Consequently, compilers must make *conservative assumptions* leading to suboptimal program performance. The potential performance degradation is especially high if the compiler's decision making chooses between transformation variants whose performance differs substantially. This is the case in distributed architectures and SDSM systems, which exhibit a large number of parameters that need to be tuned.

In ongoing work, we will build on a prototype of a dynamically adaptive compilation system [11, 10], called ADAPT. The ADAPT system can dynamically compile program variants, substitute them at runtime, and evaluate them in the executing application.

3 Extensions to OpenMP

The discussed compiler techniques can frequently generate parallel programs matching the performance of manually written message-passing programs. However, we expect that in many other cases, compiler-optimized performance will be limited. For example, with current technology the compiler may be unable to determine the best possible distribution of an array or the exact region of a distributed array that must be copied between threads before and after a parallel loop or parallel section.

We believe that programmer should be able to use the same programming paradigm when taking advantage of compiler-optimized performance and when tuning this performance. This is important because the programmer may be unable to determine whether the compiler will succeed until the program has been written. With state-of-the art programming support, programmers face the possibility of having to start from scratch in this situation. Therefore they may prefer to program entirely in a message-passing paradigm, which is guaranteed to succeed, although harder to use than a shared memory paradigm.

The solution we propose is to extend OpenMP so that programmers would be able to override the compiler whenever necessary. The extended directives will be read by the compiler and the information they contain will be used internally to guide code generation instead of the information gathered by the compiler analysis algorithms. We envision extensions to specify data distribution, computation distribution, and communication operations. One important advantage of specifying communication and distribution using directives is that, in this way, the compiler will be able to check for consistency. Communication errors are typically not detected by today's compilers, which treat MPI message passing operations as calls to library routines and do not do any semantic analysis of the actual parameters.

For data distribution, the notation used by HPF data alignment should usually suffice for arrays accessed following a regular pattern throughout the program. We expect irregularly accessed array to be handled by the SDSM mechanism.

A simple way to specify the location where a certain computation will be performed is to introduce a new clause to specify where each iteration of a `parallel do` or each `section` within a `sections` directive will execute. The clause `schedule(`*`k`*`)` could be used to specify that the section will be executed by thread *k*, which could be any expression involving program variables. In the case of a parallel loop, a clause of the form `schedule(`*`f(p)`*`)`, where *p* is the index of the parallel loop, specify that iteration *p* will be executed by thread number *f(p)*. The expression on the right-hand side of the clause could include the `home` function. Given an element `a(i,j...)`, of a distributed array, the value of `home(a(i,j, ...))` is the thread where the element has been assigned by a distribution directive.

The first version of our OpenMP extensions will only allow explicit communications outside parallel regions. Thus, distributed data accessed in the parallel region could only

```
   Q=omp_get_num_threads()
!$omp parallel do distributed(B), shared(A),
!$omp+ schedule(home(B(i))),
!$omp+ copy(1,B(UU),1)
   do i = 2,N
      A(i) = B(i) + B(i-1)
   enddo
```

Figure 4. example of the copy clause

be communicated before or after the region. In the simplest case, the data will be communicated just before or just after the region, but performing the communication at other points of the program to enable prefetching will also be possible. All the communication associated with a parallel region will be specified by a `copy` clause within the directive that marks the beginning of the region.

An example of how the `copy` directive may look like is shown in Figure 4. It is assumed in the Figure that vector `B` is `BLOCK` distributed and that the intrinsic function `UU` returns the number of the last element of `B` allocated on the current thread by the `BLOCK` distribution. The first parameter of the `copy` clause specifies that data is to be transferred from thread *p* to thread *p+1*, the second parameter is the beginning of the array region to be transferred, and the third parameter is the length of the region. The `copy` clause makes sure that element `B(i-1)` is available for the first iteration `i` executed by all threads except for thread 0.

It will be possible to associate a label with the `copy` clause and, in a separate directive, specify that the copy operation is to be performed at a certain point of the program before execution of the parallel loop starts.

Clearly, the extended OpenMP version of the loop is significantly more complex than originally and, for that reason, more error-prone for the programmer to write. This is the price that one has to pay when the compiler is not powerful enough to do the analysis and perform the appropriate transformations. Enhancing the compiler's abilities and thus reducing this price is an important goal of the work presented in this paper.

4 Conclusions

We have presented a number of compiler techniques that can be used to translate OpenMP programs for execution on distributed memory systems. Many of these techniques, which target the hybrid communication model, have already been implemented. Results obtained in preliminary experiments indicate that using the hybrid communication model often produces significant performance gains. A new OpenMP prototype compiler targeted at distributed memory machines is now under development.With this compiler and with small extensions of OpenMP we expect that the question posed in the title can be answered affirmatively.

5 Acknowledgments

The work reported in this paper was supported by NSF contract EIA 01-03582 (Purdue) and EIA 01-03610 (Illinois)

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programmig in OpenMP*. Morgan Kaufmann Publshers, 2001.

[4] H. P. F. Forum. High Performance Fortran Language Specifi cation, Version 1.0 . *Scientifi c Programming*, 2(1 & 2), 1993.

[5] J. Hoeflinger and Y. Paek. A Comparative Analysis of Dependence Testing Mechanisms. In *Thirteenth Workshop on Languages and Compilers for Parallel Computing*, August 2000.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard* , 1995.

[7] Y. Paek, J. Hoeflinger, and D. Padua. Simplication of Array Access Patterns for Compiler Optimizations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[8] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proc. of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, July 1995.

[9] M. Voss and R. Eigenmann. Dynamically adaptive parallel programs. In *International Symposium on High Performance Computing*, pages 109–120, Kyoto, Japan, May 1999.

[10] M. J. Voss and R. Eigenmann. A framework for remote dynamic program optimization. In *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00, held in conjunction with POPL'00)*, Jan. 1999.

[11] M. J. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 93 – 102. ACM Press, June 2001.

[12] J. Zhu, J. Hoeflinger, and D. Padua. Compiling for a Hybrid Programming Model Using the LMAD Representation . In *Fourteenth Workshop on Languages and Compilers for Parallel Computing*, August 2001.