An Integrated Runtime and Compile-time Approach for Parallelizing Structured and Block Structured Applications

Gagan Agrawal
Alan Sussman
Joel Saltz

CRPC-TR93368
October 1993

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

i

An Integrated Runtime and Compile-time Approach for Parallelizing Structured and Block Structured Applications

Gagan Agrawal, Alan Sussman [1]
Department of Computer Science, University of Maryland, College Park, MD 20742
Joel Saltz
UMIACS and Dept. of Computer Sc., University of Maryland, College Park, MD 20742
{gagan, als, saltz}@cs.umd.edu

Abstract

Scientific and engineering applications often involve structured meshes. These meshes may be nested (for multigrid codes) and/or irregularly coupled (called multiblock or irregularly coupled regular mesh problems). In this paper, we present a combined runtime and compile-time approach for parallelizing these applications on distributed memory parallel machines in an efficient and machine-independent fashion. We have designed and implemented a runtime library which can be used to port these applications on distributed memory machines. The library is currently implemented on several different systems. To further ease the task of application programmers, we have developed methods for integrating this runtime library with compilers for HPF-like parallel programming languages. We discuss how we have integrated this runtime library with the Fortran 90D compiler being developed at Syracuse University. We present experimental results to demonstrate the efficacy of our approach. We have experimented with a multiblock Navier-Stokes solver template and a multigrid code. Our experimental results show that our primitives have low runtime communication overheads. Further, the compiler parallelized codes perform within 20% of the code parallelized by manually inserting calls to the runtime library.

1 Introduction

In recent years, distributed memory parallel machines have been widely recognized as the most likely means of achieving scalable high performance computing. However, there are two major reasons for their lack of popularity among developers of scientific and engineering applications. First, it is very difficult to parallelize application programs on these machines. Second, it is not easy to get good speed-ups and efficiency on communication intensive applications. Current distributed memory machines have good communication bandwidths, but they also have high startup latencies which often result in high communication overheads.

Recently there have been major efforts in developing programming language and compiler support for distributed memory machines. Based on the initial works of projects like Fortran D [13, 22] and Vienna Fortran [6, 41] , the High Performance Fortran Forum has recently proposed the first version of High Performance

[1]This work was supported by ARPA under contract No. NAG-1-1485, by NSF under grant No. ASC 9213821 and by ONR under contract No. SC 292-1-22913. Research was also supported in part by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-19480 and NAS1-18605 while the authors Sussman and Saltz were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681. The authors assume all responsibility for the contents of the paper.

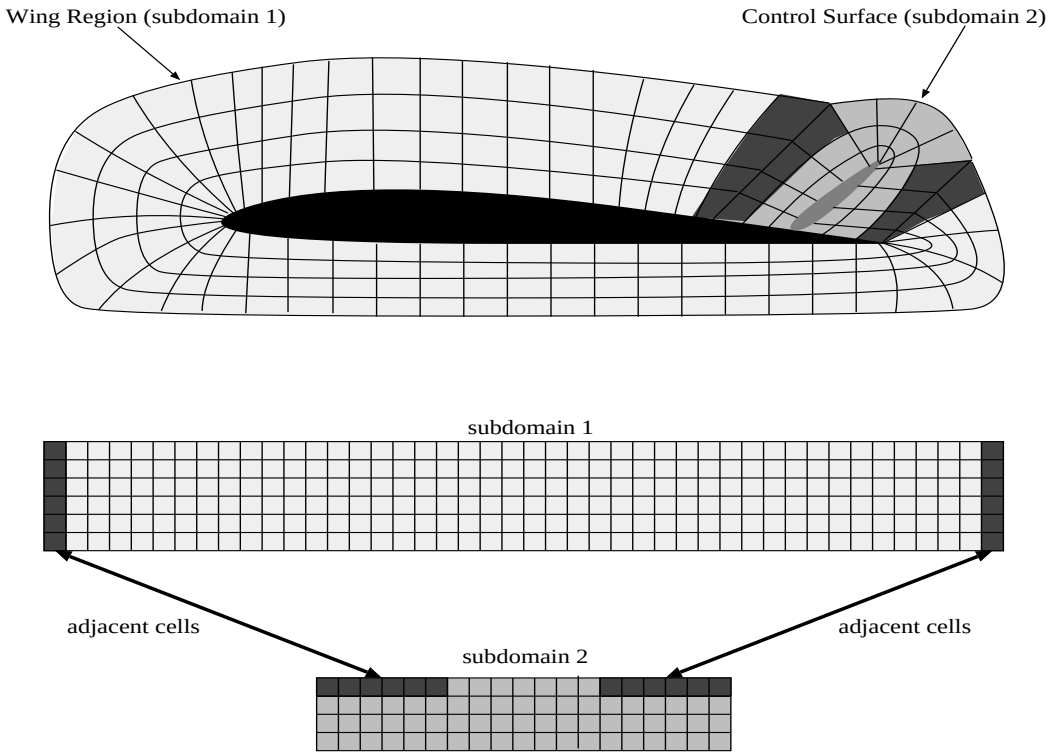


Figure 1: Block structured grid around a wing, showing an interface between blocks

Fortran (HPF) [12]. HPF allows programmer to specify the layout of distributed data and specify parallelism through operations on array sections and through parallel loops. Proposed HPF compilers are being designed to produce Single Program Multiple Data (SPMD) Fortran 77 (F77) code with message passing and/or runtime communication primitives. HPF offers the promise of significantly easing the task of programming distributed memory machines and making programs independent of a single machine architecture.

Reducing communication costs is crucial in achieving good performance on applications [20, 21]. While current systems like the Fortran D project [22] and the Vienna Fortran Compilation system [6] have implemented a number of optimizations for reducing communication costs (like message blocking, collective communication, message coalescing and aggregation), these optimizations have been developed only in the context of regular problems (i.e. in code having only regular data access patterns). Special effort is required in developing compiler and runtime support for applications that do not necessarily have regular data access patterns. Our group has already developed compiler embedded runtime support for completely irregular computations (i.e. codes in which distributed arrays are accessed based on indirection arrays) [10, 11, 18].

One class of scientific and engineering applications involves structured meshes. These meshes may be nested (as in multigrid codes) or may be irregularly coupled (called Multiblock or Irregularly Coupled Regular Mesh Problems) [9]. Multigrid is a common technique for accelerating the solution of partial-differential equations [5, 30]. Multigrid codes employ a number of meshes at different levels of resolution. The restriction and prolongation operations for shifting between different multigrid levels require moving regular array sections [19] with non-unit strides. In multiblock problems, the data is divided into several interacting regions (called blocks or subdomains). There are computational phases in which regular computation is performed on each block independently. Boundary updates require communication between blocks, which is restricted to moving regular array sections (possibly including non-unit strides).

Multiblock grids are frequently used for modeling geometrically complex objects which cannot be easily

modeled using a single regular mesh [2, 3, 24, 29, 36]. In Figure 1, we show how the area around an aircraft wing has been modeled with a multiblock grid. Multiblock applications are used in important grand-challenge applications like air quality modeling [28, 32], computational fluid dynamics [40], structure and galaxy formation [25, 38], simulation of high performance aircrafts [1, 8, 31], large scale climate modeling [14], reservoir modeling for porous media [14], simulation of propulsion systems [14], computational combustion dynamics [14], geophysical databases [33], and land cover dynamics [23].

In this paper we present a combined runtime and compile-time approach for parallelizing this general class of applications on distributed memory machines. We present runtime support that we have designed and implemented for parallelizing these applications on distributed memory machines in an efficient, convenient and machine independent manner. We state the extensions required to the current version of HPF for handling block structured codes. We present methods by which the compilers for HPF style parallel programming languages can automatically generate calls to our runtime primitives. We discuss how we have integrated our runtime primitives with the Fortran 90D compiler being developed at Syracuse University [4]. While the design of our runtime system was initially motivated by multigrid and multiblock applications, our primitives can also be used in many cases for parallelizing computations with regular data access patterns.

We present experimental results to demonstrate the efficacy of our approach. We have experimented with one multiblock application [40] and one multigrid code [35]. We have measured the runtime overheads of our primitives. We have compared the performance of compiler parallelized multiblock and multigrid templates with those of the hand parallelized (i.e. parallelized by inserting calls to the runtime library by hand) versions. Our experimental results show that the primitives have low runtime communication overheads and the compiler parallelized codes performs within 20% of the codes parallelized by inserting calls to runtime library by hand. We discuss the optimizations that we have used to achieve this performance.

Several other researchers have also developed runtime libraries or programming environments for multiblock applications. Baden [24] has developed a Lattice Programming Model (LPAR). This system, however, achieves only coarse grained parallelism since a single block can only be assigned to one processor. Quinlan [26] has developed P++, a set of C++ libraries for grid applications. While this library provides a convenient interface, the libraries do not optimize communication overheads. Our library, on the contrast, reduces communication costs by using message aggregation.

The rest of this paper is organized as follows. In Section 2, we introduce the runtime library that we have developed. In Section 3, we discuss the regular section analysis required for efficiently generating schedules for regular section moves, one of the communication primitives in our library. In Section 4, we state the extensions required to the current version of HPF and discuss how the compiler recognizes the data access patterns which can be handled using the runtime primitives that we have developed. In this section we also discuss the compiler transformations for automatically inserting the calls to the runtime library routines. In Section 5, we present experimental results to study the communication overheads of our primitives and the effectiveness of the compiler. We conclude in Section 6.

2 Runtime Support

In this section we present the details of the runtime support library that we have designed for parallelizing multiblock and multigrid codes on distributed memory machines. We discuss the nature of computation and communication in multiblock and multigrid codes and also describe how the library primitives facilitate parallelization of these applications.

The set of runtime routines that we have developed is called the Multiblock Parti library [39]. In summary, these primitives allow an application programmer or a compiler to

- Lay out distributed data in a flexible way, to enable good load balancing and minimize interprocessor communication,
- Give high level specifications for performing data movement, and
- Distribute the computation across the processors.

We have designed the primitives so that communication overheads are significantly reduced (by using message aggregation). These primitives provide a machine-independent interface to the compiler writer and applications programmer. We view these primitives as forming a portion of a portable, compiler independent, runtime support library.

This library is currently implemented on the Intel iPSC/860, the Thinking Machines' CM-5 and the PVM message passing environment for network of workstations [15]. The design of the library is architecture independent and therefore it can be easily ported on any distributed memory parallel machine or any environment which supports message passing (e.g. Express). The library primitives can currently be invoked from Fortran or C programs. Programmers can port their Fortran or C programs on distributed memory machines by manually inserting calls to the library routines. The resulting program has Single Program Multiple Data (SPMD) model of parallelism.

While the design of our runtime system was initially motivated by multigrid and multiblock applications, our runtime primitives are also applicable in many cases for regular codes. In Section 4, we briefly mention other cases when our primitives can be used. In this section, however, we discuss the details of our runtime support in context of multiblock and multigrid applications.

2.1 Multiblock and Multigrid Applications

For a typical multiblock application, the main body of the program consists of an outer sequential (time step) loop, and an inner parallel loop. The inner loop iterates over the blocks of the problem, after applying boundary conditions to all the blocks (including updating interfaces between blocks). Applying the boundary conditions involves interaction (communication) between the blocks. In the inner loop over the blocks, the computation in each block is typically sweeps over mesh in which each mesh-point interacts only with its nearby neighbors. Since in these applications, there are computational phases which involve interactions only within each block, communication overheads are reduced if each block is not divided across a large number of processors. So, blocks may have to be distributed onto subsets of processor space. Since the number of blocks is typically quite small (i.e. at most a few dozens), at least some of the blocks will have to be distributed across multiple processors. Partitioning of the parallel loop across the block is the source of the coarse-grained parallelism for the application. Furthermore, within each iteration of the inner loop fine-grained parallelism is available in the form of (large) parallel loops, iterating over the elements of the blocks.

Now, we briefly discuss the runtime support required for parallelizing multiblock applications. First, there must be a means for expressing data layout and organization on the processors of the distributed memory parallel machine. We need compiler and runtime support for mapping blocks (arrays) to subsets of the processor space. Second, there must be methods for specifying the movement of data required. Two types of communication are required in multiblock applications. The interaction between different blocks requires the movement of regular array sections. The inner loop involves interactions among neighboring elements of the grids. Since blocks may be partitioned across processors, this also requires communication. Third, there must be some way of distributing loops iterations among the processors and converting global distributed arrays references to local references.

Multigrid is a common technique for accelerating the solution of partial differential equations. Multigrid codes employ a number of meshes at different levels of resolution. Multigrid codes have phases of restriction

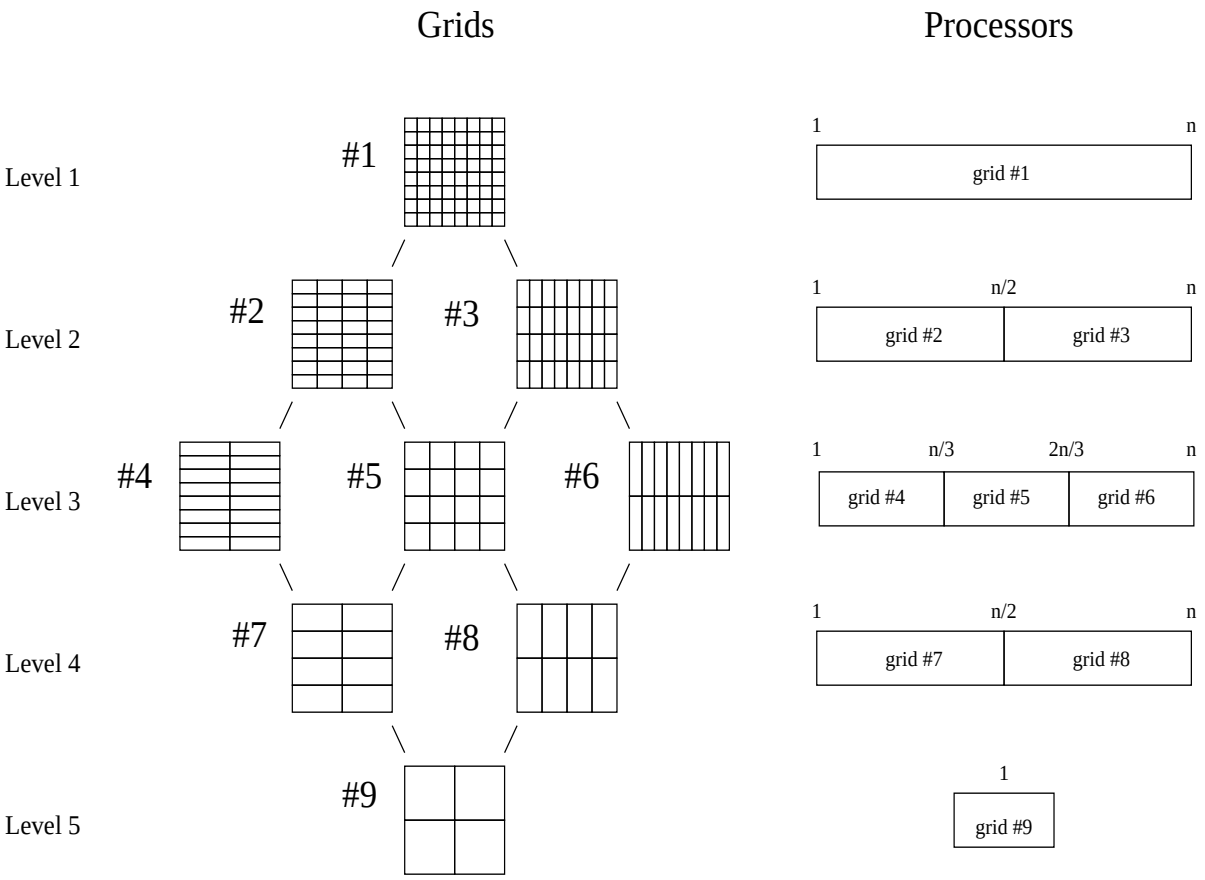


Figure 2: Semi-coarsened grids and their mapping to processors

(in which a coarse grid is initialized based upon a finer grid), prolongation (in which a coarse grid is copied into a finer grid with non-unit stride and then the other elements on the fine grid are computed by interpolation) and sweeps over individual grids. Coarse meshes may be obtained from fine grid by coarsening by the same factor or different factors along different dimensions. Accordingly, each grid may be distributed over the entire set of processors, or some grids may have to be distributed over parts of the processor space. A particular form of multigrid technique is the semi-coarsening multigrid technique [34]. Semi-coarsening multigrid works as follows. Starting from the finest grid, coarser grids are generated by coarsening by different factors along different dimensions. There may be many grids, having the same number of mesh points, but obtained by different coarsening factor along each dimensions (i.e. they may have different shapes). In parallelizing such an application, communication costs can be reduced while maintaining good load balance by the following mapping scheme. The finest grid is mapped over the entire processor space. Different grids at the same level (i.e. having the same total number of mesh points but obtained by different coarsening factors along each dimension) are mapped to disjoint parts of the processor space. In Figure 2 we show how the semi-coarsened grids are generated and how they can be mapped to the processor space.

The runtime support requirements for the multigrid codes is as follows. As with multiblock codes, we need to be able to map grids over subsets of the processor space. The restriction and prolongation steps require regular section moves between grids at different levels of resolution. Again, communication arises because each grid may be distributed across multiple processors and computation within each block requires near neighbor interactions. Similarly, the interpolation required during the prolongation step also involves interaction among neighboring elements. Also, support for distributing loop iterations and transforming global distributed array

references to local references is required.

2.2 Multiblock Parti Primitives

We now discuss the design of our runtime library [39]. Since, in typical multiblock and multigrid applications, the number of blocks and their respective sizes is not known until runtime, the distribution of blocks onto processors is done at runtime. The distributed array descriptors (DAD) [4] for the arrays representing these blocks are, therefore, generated at runtime. Distributed array descriptors contain information about the portions of the arrays residing on each processor, and are used at runtime for performing communication and distributing loops iterations. We will not discuss the details of the primitives which allow the user to specify data distribution. For more details, see [39].

As we discussed previously, two types of communication are required in both multiblock and multigrid applications. Inter-block communication is required because of boundary conditions between blocks (in multiblock codes) and restrictions and prolongations between grids at different levels of resolution (in multigrid codes). Since the data that needs to be communicated is always a regular section of an array, this can be handled by primitives for regular section move. A regular section move copies a regular section of one distributed array into regular section of another distributed array, potentially involving changes of offset, stride and index permutation. Intra-block communication is required because of partitioning of blocks or grids across processors. The data access pattern in the computation within a block or grid is regular. This implies that the interaction between grid points is restricted to nearby neighbors. The interpolation required during the prolongation step in multigrid codes also involves interaction among the neighboring array elements. Such communication is handled by allocation of extra space at the beginning and end of each array dimension on each processor. These extra elements are called *overlap*, or *ghost*, cells [6, 16, 27]. Depending upon the data access pattern in a loop, the required data is copied from other processors and is stored in the overlap cells.

In our runtime system, communication is performed in two phases. First, a subroutine is called to build a communication *schedule* that describes the required data motion, and then another subroutine is called to perform the data motion (sends and receives on a distributed memory parallel machine) using a previously built schedule. Such an arrangement allows a schedule to be be used multiple times in an iterative algorithm. The communication primitives include a procedure *Overlap_Cell_Fill_Sched*, which computes a schedule that is used to direct the filling of overlap cells along a given dimension of a distributed array. Communication for filling in the *overlap* cells has been implemented in other systems for regular computations [6, 16, 27], so we will not be discussing the details here. The primitive *Regular_Section_Copy_Sched* carries out the preprocessing required for performing the regular section moves. In section 3, we discuss the details of the regular section analysis required for efficiently generating the schedule for regular section move.

The schedules produced by *Overlap_Cell_Fill_Sched* and *Regular_Section_Copy_Sched* are employed by a primitive called *Data_Move* that carries out both interprocessor communication (sends and receives) and intra-processor data copying.

The final form of support provided by the multiblock Parti library is to distribute loop iterations and transform global distributed arrays references into local references. Our library distributes loop iterations based upon the owners compute rule, which means that a particular loop iteration is executed by the processor owning the left-hand side array element written into during that iteration. As we discussed earlier, we prefer to generate the Distributed Array Descriptor (DAD) for the arrays at runtime. This means that global indices can be translated to local indices only at runtime and not at compile-time. Two primitives, *Local_Lower_Bound* and *Local_Upper_Bound*, are provided for transforming loop bounds (returning, respectively, the local lower and upper bounds of a given dimension of the referenced distributed array) based upon the owners compute rule. Primitives *global_to_local* and *local_to_global* are also available for translating a global index into local

index and translating a local index on a processor into global index respectively.

3 Regular Section Analysis

In this section we discuss the regular section analysis required for efficiently generating schedules for regular section moves (i.e. for implementing the primitive *Regular_Section_Copy_Sched*). By regular section analysis we mean how each processor can determine, for each other processor, the exact parts of the distributed array it needs to send and receive, given the source and the destination regular sections in global coordinates. In our current system, this analysis is always done at runtime. However, if the distributions of source and destination distributeed arrays and description of source and destination regular sections are available at compile-time, then this analysis can be done at compile-time as well. In separate works, Chatterjee *et al* [7], Stichnoth [37] and Gupta *et al* [17] have developed compile-time methods for analyzing and generating communication associated with HPF's forall statements and/or F90 style array expressions. While their solutions work for the general case when the data distributions are block-cyclic, their methods require that the data distribution be known at the compile-time and the exact description of the statement be available in terms of compile-time constants. We are interested in techniques which can be used at runtime and are specialized towards the particular communication patterns associated with multigrid and multiblock applications. Also, since these applications typically need block distribution, we restrict to describing our analysis when the data is block distributed along each dimension.

We assume that the array indices always start from 0 (as in the C programming language). The processors owning source (or, destination) array can be viewed as forming an r-dimensional virtual processor grid. A processor p in this processor grid has coordinates $\{p_1, p_2, \ldots, p_r\}$. We assume that the numbering starts from zero in each dimension in the processor grid.

The source regular section, denoted by $\mathcal{S}$, is part of the source distributed array s.

$$\mathcal{S} = \{(s\_lo_1 : s\_hi_1 : s\_str_1), (s\_lo_2 : s\_hi_2 : s\_str_2), \ldots, (s\_lo_r : s\_hi_r : s\_str_r)\}$$

s_lo_i, s_hi_i and s_str_i are respectively the lowest index, highest index and the stride along the i^{th} dimension (in global indices). The regular section $\mathcal{S}$ defines a set of array elements. An array index e_i along the i^{th} dimension is said to be a part of the regular section iff $\exists l_i$ (an integer) s.t.

$$e_i \quad = \quad s\_lo_i + l_i \cdot s\_str_i, \tag{3.0.1}$$

where,

$$0 \ \leq \ l_i \ \leq \frac{s\_hi_i - s\_lo_i}{s\_str_i}$$

An array element whose indices are $(e_1, e_2, \ldots, e_r)$ belongs to the regular section $\mathcal{S}$ iff,
$\forall\, i,\ 1 \leq i \leq r,$ the array index e_i along the i^{th} dimension belongs to the regular section $\mathcal{S}$.

We will use this format to describe all regular sections. The destination regular section, denoted by $\mathcal{D}$, is a part of the destination array d.

$$\mathcal{D} = \{(d\_lo_1 : d\_hi_1 : d\_str_1), (d\_lo_2 : d\_hi_2 : d\_str_2), \ldots, (d\_lo_r : d\_hi_r : d\_str_r)\}$$

Regular section moves can involve index permutations. We denote by $im(i)$ the destination dimension which is accessed by the same loop index as the i^{th} source dimension.

For each processor owning part of the source regular section, we want to determine the set of local elements that it will be sending to each processor owning part of the destination regular section. We call these sets of

elements *send sets*. Similarly, for each processor owning part of the destination, we want to determine the set of local elements that it will be receiving from each processor owning part of the source. We call these sets of elements *receive sets*. Here we just discuss the analysis that a particular source processor does to compute the *send sets*. The analysis for determining the *receive sets* is completely analogous and is therefore not described.

The steps we follow for computing the *send sets* are as follows. For a processor p, we determine the part of regular section $\mathcal{S}$ that it owns, that is, we restrict the section $\mathcal{S}$ on the processor p (which is denoted by $\mathcal{S}'(p)$). Next, we take a transformation of the section $\mathcal{S}'(p)$) to map it from the source regular section $\mathcal{S}$ to the destination regular section $\mathcal{D}$. The resulting section is denoted by $\mathcal{D}'(p)$. We next determine the set of destination processors which own part of the section $\mathcal{D}'(p)$, i.e. the destination processors to whom the processor p will be communicating. For each such processor q, we restrict the section $\mathcal{D}'(p)$ to determine the part that q owns, calling it $\mathcal{D}''(p,q)$. In the last step, the section $\mathcal{D}''(p,q)$ is mapped back to the source, the resulting section is denoted by $\mathcal{S}''(p,q)$. $\mathcal{S}''(p,q)$ is the *send set* that the source processor p will be sending to the destination processor q.

We now present the details of the steps mentioned above. We consider a particular processor p which owns part of the source array s, of which the regular section $\mathcal{S}$ is a part. Let $llo_i^S(p)$ and $lhi_i^S(p)$ be the lowest and the highest points along the i^{th} dimension (in global indices) that the processor p owns. Since the data distribution is block, the processor p owns a contiguous chunk of data from $llo_i^S(p)$ to $lhi_i^S(p)$.

3.1 Restricting $\mathcal{S}$ to the Processor p

We now compute the part of the regular section $\mathcal{S}$ that the processor p owns. This is denoted by $\mathcal{S}'(p)$. Through-out our discussion, all regular sections will always be described in global indices. Given the global coordinates, any individual processor can always determine the corresponding local (on processor) indices.

$$\mathcal{S}'(p) = \{(s\_lo'_1(p), s\_hi'_1(p), s\_str'_1(p)), \ldots, (s\_lo'_r(p), s\_hi'_r(p), s\_str'_r(p))\}$$

where,

$$s\_lo'_i(p) = s\_lo_i + \left\lceil \frac{max(0, llo_i^S(p) - s\_lo_i)}{s\_str_i} \right\rceil \cdot s\_str_i \quad (3.1.1)$$

$$s\_hi'_i(p) = min(lhi_i^S(p), s\_hi_i) \quad (3.1.2)$$

$$s\_str'_i(p) = s\_str_i \quad (3.1.3)$$

Since the data is block distributed, $s\_str'_i(p)$ is s_str_i. $s\_lo'_i(p)$ is the first index along the i^{th} dimension which is part of the regular section $\mathcal{S}$ and is owned by the processor p. In the calculation of $s\_lo'_i(p)$ there are two cases, depending upon whether $s\_lo_i \geq llo_i^S(p)$ or $s\_lo_i < llo_i^S(p)$. If $s\_lo_i \geq llo_i^S(p)$ then the index s_lo_i is on the processor p. Therefore, $s\_lo'_i(p) = s\_lo_i$. Alternatively, if $s\_lo_i < llo_i^S(p)$, then the expression for $s\_lo'_i(p)$ reduces to $s\_lo_i + \lceil (llo_i^S(p) - s\_lo_i)/s\_str_i \rceil \cdot s\_str_i$. This is the first index after $llo_i^S(p)$ which is part of the regular section. Note that $s\_hi'_i(p)$ is not necessarily a member of the regular section $\mathcal{S}$. It just needs to be greater than the last member of regular section on the processor q, so that the loop accessing successive indices in the section terminates correctly.

If the processor p does not own any part of the regular section along the i^{th} dimension, then the above expressions will give a value of $s\_lo'_i(p)$ which is greater than the value of $s\_hi'_i(p)$. In general, if

$$\exists i,\ 1 \leq i \leq r, \quad s.t.\ s\_lo'_i(p) > s\_hi'_i(p)$$

then the processor p does not own any part of the regular section $\mathcal{S}$.

3.2 Mapping $\mathcal{S}'(p)$ to the Destination

Next, we determine the corresponding section in the destination array (i.e. part of the destination regular section that will be received from the processor p). This is denoted by $\mathcal{D}'(p)$.

$$\mathcal{D}'(p) = \{(d\_lo'_1(p), d\_hi'_1(p), d\_str'_1(p)), \ldots, (d\_lo'_r(p), d\_hi'_r(p), d\_str'_r(p))\}$$

where,

$$j = im(i) \tag{3.2.1}$$

$$d\_lo'_j(p) = d\_lo_j + \frac{s\_lo'_i(p) - s\_lo_i}{s\_str_i} \cdot d\_str_j \tag{3.2.2}$$

$$d\_hi'_j(p) = d\_lo_j + \left\lfloor \frac{s\_hi'_i(p) - s\_lo_i}{s\_str_i} \right\rfloor \cdot d\_str_j \tag{3.2.3}$$

$$d\_str'_j(p) = d\_str_j \tag{3.2.4}$$

Since the array is distributed by blocks, $d\_str'_j(p)$ is d_str_j. The expressions for $d\_lo'_j(p)$ and $d\_hi'_j(p)$ follow from finding the indices in the destination regular section which correspond to the indices $s\_lo'_i(p)$ and $s\_hi'_i(p)$, respectively, in the source regular section.

3.3 Restricting $\mathcal{D}'(p)$ to Processor q

We first determine the set of processors to which the source processor p will send data. The processors owning parts of the destination array form an r-dimensional virtual processor grid. A processor q, owning part of the destination array, has coordinates $\{q_1, q_2, \ldots, q_r\}$ in this processor grid. We assume that each processor owns $size_i$ indices along the i^{th} dimension.

We denote by $q\_min_i(p)$ and $q\_max_i(p)$ the lowest and highest coordinates along the i^{th} dimension of the processors which own part of the regular section $\mathcal{D}'(p)$.

$$q\_min_i(p) = \left\lceil \frac{d\_lo'_i(p) + 1}{size_i} \right\rceil - 1 \tag{3.3.1}$$

$$q\_max_i(p) = \left\lceil \frac{d\_hi'_i(p) + 1}{size_i} \right\rceil - 1 \tag{3.3.2}$$

A processor q having coordinates $\{q_1, q_2, \ldots, q_r\}$ will receive data from the source processor p iff

$$\forall i,\ 1 \le i \le r,\ \ q\_min_i(p) \le q_i \le q\_max_i(p)$$

Consider a particular processor q which will receive data from the source processor p. Suppose that the start and end points along the i^{th} dimension on this processor are $llo_i^d(q)$ and $lhi_i^d(q)$ respectively. We denote the part of the destination regular section that the processor q will receive from the processor p by $\mathcal{D}''(p,q)$.

$$\mathcal{D}''(p,q) = \{(d\_lo''_1(p,q), d\_hi''_1(p,q), d\_str''_1(p,q)), \ldots, (d\_lo''_r(p,q), d\_hi''_r(p,q), d\_str''_r(p,q))\}$$

where,

$$d\_lo''_i(p,q) = d\_lo'_i + \left\lceil \frac{max(0, llo_i^d(q) - d\_lo'_i(p))}{d\_str_i} \right\rceil \cdot d\_str_i \tag{3.3.3}$$

$$d\_hi''_i(p,q) = min(lhi_i^d(q), d\_hi'_i(p)) \tag{3.3.4}$$

$$d\_str''_i(p,q) = d\_str_i \tag{3.3.5}$$

The reasoning behind the correctness of the above expressions is the same as that used in determining $\mathcal{S}'$ from $\mathcal{S}$, as discussed in Section 3.1.

3.4 Mapping $\mathcal{D}''(p,q)$ to the Source

Next, we determine the equivalent part of the regular section $\mathcal{D}''(p,q)$ on the source side (i.e. the part of the source regular section which the processor p sends to the processor q). We denote this by $\mathcal{S}''(p,q)$.

$$\mathcal{S}''(p,q) = \{(s\_lo_1''(p,q), s\_hi_1''(p,q), s\_str_1''(p,q)), \ldots, (s\_lo_r''(p,q), s\_hi_r''(p,q), s\_str_r''(p,q))\}$$

where,

$$j = im(i) \tag{3.4.1}$$

$$s\_lo_i''(p,q) = s\_lo_i + \frac{d\_lo_j''(p,q) - d\_lo_j}{d\_str_j} \cdot s\_str_i \tag{3.4.2}$$

$$s\_hi_i''(p,q) = s\_lo_i + \left\lfloor \frac{d\_hi_j''(p,q) - d\_lo_j}{d\_str_j} \right\rfloor \cdot s\_str_i \tag{3.4.3}$$

$$s\_str_i''(p,q) = s\_str_i \tag{3.4.4}$$

The reasoning behind the correctness of the these expressions is the same as that used in determining $\mathcal{D}'(p)$ from $\mathcal{S}'(p)$ in Section 3.2.

3.5 Discussion

All the calculations described above are performed by the processor p locally and do not involve communication with any other processor. Therefore, send and receive sets can be generated efficiently. Based on the calculation of $\mathcal{S}''$, the processor p knows the contents of the message that it must send to processor q. However, when processor q receives this message, it does not have any information about which local memory locations each element of the message must be copied into. To facilitate this, each destination processor computes the set of (local) elements that it will receive from each source processor. The calculations for computing these *receive sets* are completely analogous to the computations for the *send sets*. Therefore, we do not describe the computation of the *receive sets* here. The source processor p always sends the set of elements it needs to send to the processor q in a single message, packed in the column major fashion. Processor q can then use the receive set information to copy the elements in the received message into the appropriate local elements.

An alternative to this scheme is that the message sent by the source processor p also contains information about what local memory location at the destination processor q each of elements packed in the message needs to be copies to. The destination processor q can then copy elements of the message into its local elements based upon this information. This approach does save some computation at the destination processors. However, the size of the messages increases significantly because of the extra information that needs to be sent. In our implementation, we have chosen to compute both the send and receive sets, since on current distributed memory machines, this is less expensive than communicating the receive set information.

3.6 Example

Consider a regular section move that involves a source array of size $100 * 100$ and a destination array of size $50 * 100$. The source array is block distributed over a $2 * 2$ virtual processor grid and the destination array is block distributed over a $4 * 1$ virtual processor grid. The source and the destination regular sections are: $\mathcal{S} = \{(10:60:2), (10:70:3)\}$ and, $\mathcal{D} = \{(10:30:1), (5:80:3)\}$. The first dimension of the source regular section is aligned to the second dimension of the destination regular section and the second dimension of the source regular section is aligned to the first dimension of the destination regular section i.e. $im(1) = 2$ and $im(2) = 1$.

We consider the source processor p with coordinates $\{1, 0\}$. The part of the global array that this processor owns is $llo_1^S(p) = 50$, $lhi_1^S(p) = 99$, $llo_2^S(p) = 0$ and $lhi_2^S(p) = 49$.

The part of the source regular section that processor p owns ($\mathcal{S}'(p)$) is given by

$$\mathcal{S}'(p) = \{(50 : 60 : 2), (10 : 49 : 3)\}$$

The corresponding section on the destination side ($\mathcal{D}'(p)$) is given by

$$\mathcal{D}'(p) = \{(10 : 23 : 1), (65 : 80 : 3)\}$$

Next, the processor p determines the set of destination processors with whom it will be communicating. We have for the destination array, $size_1 = 50$ and $size_2 = 25$.

This gives, $p\_min_1(p) = 0$, $p\_max_1(p) = 0$, $p\_min_2(p) = 2$, and $p\_max_2(p) = 3$. The destination processors the source processor p will communicate with are the ones with grid coordinates $\{0, 2\}$ and $\{0, 3\}$. Consider the destination processor q with coordinates $\{0, 3\}$. The part of the destination array that processor q owns is given by $llo_1^d(q) = 0$, $lhi_1^d(q) = 49$, $llo_2^d(q) = 75$ and $lhi_2^d(q) = 99$.

The part of regular section which the processor p will be sending to the processor q ($\mathcal{D}''(p, q)$) is given by

$$\mathcal{D}''(p, q) = \{(10 : 23 : 1), (77 : 80 : 3)\}$$

The corresponding source section ($\mathcal{S}''(p, q)$) is now given as

$$S''(p, q) = \{(58 : 60 : 2), (10 : 49 : 3)\}$$

4 Compiler Support

In this section we first discuss the additional functionality required in the current version of HPF to support multiblock and multigrid codes. We describe how a compiler can analyze the data access patterns associated with a loop, to recognize communication patterns which can be handled using the runtime primitives for multiblock problems. We then describe the compiler transformations for generating the calls to these runtime primitives. We also briefly discuss how loop iterations are distributed to achieve parallelism.

4.1 Language Support

The current version of HPF does not support all the functionality required for multiblock and multigrid applications. In multiblock problems, the problem geometry is divided into a number of blocks of different sizes. As we have discussed in the previous sections, each of these blocks needs to be distributed onto a portion of the processor space. Similarly, in multigrid codes, communication overheads can typically be reduced by distributing each coarse grid over a part of the processor space [35]. The current version of HPF does not provide any convenient mechanism for distributing arrays (or templates) onto a part of the processor space. We therefore need additional functionality for conveniently distributing arrays onto part of the processor space. In HPF, the programmer declares an abstract processor space by using the processor directive:

```
!HPF$ PROCESSORS P(N)
```

In general, the abstract processor space can have any number of dimensions. To support block structured applications, we need to be able to specify processor subspaces. We declare a processor subspace as follows:

```
!HPF$ PSUBSPACE P1 IS P(LB:UB)
```

The above directive states that P1 is the part of the processor space P which starts at processor LB and ends at processor UB. In addition, if the processor subspace P1 is created from the processor space P, then P1 must have the same number of dimensions as P. Since the sizes of the blocks are, in general, not known at compile-time, the subspace directive must be executable, so that the parameters do not have to be compile-time constants. Once a processor subspace has been declared, arrays or templates can be distributed onto it. For example,

```
!HPF$ TEMPLATE T(100,100)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P1
```

Similar functionality is available in Vienna Fortran [41], where a distribution can be mapped to a processor reference. We prefer to explicitly name the processor subspace since it makes it easier to detect at compile-time which arrays or templates have been mapped to the same processor subspace, even if the exact size of the subspace is not available as compile-time constants.

With the block distributions supported in the current version of HPF, the entire array gets distributed uniformly across the processors of the distributed memory parallel machine. This may not be ideal for load balancing for many applications. While the programmer may declare a large array, not all the elements of the array may be actual mesh points participating in computation. Some of the array elements at both ends of each dimension may be used for participating in exchanges between blocks. We refer to such array elements as *external ghost cells*. For example, the actual declared arrays for a given block may be $52 \times 12 \times 12$, with two external ghost cells at the beginning and end of each dimension. This means that the actual mesh representing the computation is of size $48 \times 8 \times 8$. It is these mesh points which must be distributed evenly across the processors onto which the block is distributed, so that the computation load will be evenly balanced. The external ghost cells at both ends of each dimension are then stored at the first and last processor along that dimension in the processor space. For example, if an array with 8 elements, plus two external ghost cells on each end (for a total of 12 elements), is distributed on 4 processors, we would like to store 2 mesh points on each processor along that dimension. The first and last processors can then store the external ghost cells at the beginning and end, respectively. This results in a much better load balance than simply distributing 3 array elements onto each processor (which will result in the first and last processors having only 1 real mesh point and the intermediate processors having 3 real mesh points each).

The current version of HPF does not provide any mechanism for specifying external ghost cells. We need additional functionality in the align statement to express them. We do this by explicitly specifying the number of external ghost cells at the beginning and end of each dimension:

```
!HPF$ DIMENSION A(105,105)
!HPF$ ALIGN A(i,j) WITH T(i:2:3,j:2:3)
```

This example says that an array of size 105x105 is aligned along a template of size 100x100, with 2 external ghost cells at the beginning of each dimension and 3 external ghost cells at the end of each dimension. If the template T is distributed by blocks onto a two dimensional processor space, A(3:102,3:102) also gets distributed in the same fashion. Note that our purpose here is not to introduce new syntax but to achieve the additional functionality that we need. We believe that this functionality will be added, in some form, in a future version of HPF.

4.2 Identifying Communication Patterns

In this subsection we discuss how the compiler identifies the communication patterns which can be handled using the runtime support for multiblock problems. Note that our purpose is not to provide a general framework for compiling *forall* statements; we are only interested in recognizing the patterns that can be handled efficiently using the primitives we have developed.

While we have designed the runtime support with multiblock and multigrid codes in mind, the runtime primitives can also be used to efficiently handle communication for many other types of applications. Regular section move primitives can be used for handling the communication required when a distribution of an array is changed (using the redistributed statement of HPF [12]. They can also be used to handle the communication required for filling ghost cells when the data distribution is cyclic or block-cyclic. The primitives can also be used for handling communication in forall loops and array expressions in many regular applications, especially when strides are involved.

We do not consider applications in which indirection arrays may need to be analyzed to identify communication patterns. The irregular communication arising from use of indirection arrays can be handled using the Parti primitives for irregular problems [10], which have also been integrated with compilers for HPF-style languages (including the Rice University Fortran 77D compiler [18] and the Syracuse University Fortran 90D compiler [4]). F90D and HPF also provide a number of intrinsic functions (such as reduction, spread, etc.). We assume that if a computation can be done using these intrinsics, it is either written this way by the programmer or is recognized by the compiler in an early phase of the compilation.

HPF allows multiple statement forall loops and array expressions for expressing parallelism. We restrict our discussion to the problem of analyzing a single statement *forall* loop for communication patterns. A multiple statement *forall* loop is just like a single statement *forall* loop, with the multiple assignment statements all having the same loop header. The same analysis can be done for each assignment statement within the *forall* loop. The array expressions provided by HPF can always be translated into equivalent *forall* loops.

We classify the data access patterns associated with a *forall* loop as being one of three kinds:

- Completely regular (not involving any communication).
- Ones that can be handled by filling in overlap cells.
- Ones that requires regular section moves.

Consider any *forall* statement with array expressions involving an array A in the left hand side and an array B as one of the arrays on the right hand side. The form of the *forall* statement is assumed to be as follows:

$$\begin{aligned} &\text{forall } (i_1 = lo_1 : hi_1 : st_1, \ldots, i_m = lo_m : hi_m : st_m) \\ &\qquad \mathrm{A}(f_1, f_2, \ldots, f_j) = \cdots \mathrm{B}(g_1, g_2, \ldots, g_n) \cdots \end{aligned}$$

The i_k, $(k = 1..m)$ are the loop variables associated with the *forall* statement. lo_k, hi_k and st_k are respectively the lower bound, upper bound and the stride for each loop variable. For the left hand side array A, f_1, f_2, .., f_j are the subscripts. Similarly, for the right hand side array B, g_1, g_2, .., g_n are the subscripts. The form of the array subscripts f and g is assumed to be:

$$f_k = c1_k i1_k + d1_k$$

$$g_k = c2_k i2_k + d2_k$$

Here, $i1_k$ and $i2_k$ are loop variables. If a subscript is a loop invariant scalar, then we say that the loop variable $i1_k$ (or, $i2_k$), is ϕ and $c1_k$ (or, $c2_k$) is 0. $c1_k$, $c2_k$, $d1_k$ and $d2_k$ may be expressions, but we assume

that they do not involve any loop variable. Our primitives are not applicable for cases in which multiple loop variables are associated with a particular array subscript or when the same loop variable appears in more than one subscript for a particular array or when a subscript is a higher order function of a loop variable. Such cases can be handled by using the Parti primitives for irregular problems. Also, the HPF specification allows the lower bound, upper bound and stride expressions for each loop variable to be evaluated in any order. Consequently, the lower bound, upper bound and stride for any loop variable are not allowed to be a function of any other loop variable. It is possible, in general, that a loop variable may appear only in the right hand side array or only in the left hand side array. If a particular loop variable appears only in the right hand side, this represents successive overwrites on the same memory location of the left hand side array. Such code is not likely to appear in practice and therefore, we do not consider this case. If a particular loop variable appears only in the left hand side array, this represents a spread operation. We assume that it is written using the intrinsic spread operation, and is not a part of the *forall*.

Depending upon how the arrays A and B are distributed and aligned with respect to each other, we consider three different cases. These are:

Case I: Arrays A and B are aligned to different templates.

Case II: Arrays A and B are identically aligned to the same template. This case also requires that A and B are arrays of identical shape and size, i.e. having the same number of dimensions and the same size in each dimension.

Case III: Arrays A and B are aligned to the same template, but with a different stride, offset and/or index permutation. This means that the the arrays A and B are mapped to the same processor subspace, but each in a different manner.

We now discuss how the data access pattern associated with each of these cases is analyzed. The transformations required for generating calls to the runtime primitives are discussed in Section 4.3.

4.2.1 Case I

Since the arrays are aligned to distinct templates, the communication is always handled using the regular section move primitive from the runtime library. We expect that if a user has declared distinct templates then they are either distributed over different processor subspaces, or have a different number of distributed dimensions. Therefore, there is no regularity in the communication associated with a *forall* statement containing references to such arrays.

It is possible that a programmer may create more than one template with the same number of distributed dimensions, distributed over the same processor subspace. We can extend our analysis to consider the processor subspace over which distinct templates are distributed in determining any regularity in the communication required. However, we do not discuss this possibility here.

4.2.2 Case II

The data access patterns associated with this case may be completely regular, or may require the overlap cells to filled in, or may require a regular section move.

Let DD(A) denote the set of dimensions of the array A which are distributed. Under the assumptions for Case II, DD(B) = DD(A). In terms of the form of the *forall* statement and the array subscripts that presented in Section 4.2, the condition for the communication associated with the forall to require a regular section move is :

$\exists\, j \in DD(A)$ s.t.

Arrays A and B are aligned identically

L.H.S. Expression	R.H.S. Expression	Regular Section Move Required	Overlap Cell Fill Required
A(i,j)	B(j+2,i+1)	YES	NO
A(i,j)	B(2*i,j)	YES	NO
A(i,j)	B(i+n1,j+2)	YES	NO
A(i,j)	B(i+1,j+2)	NO	YES
A(i,j)	B(i,j)	NO	NO

Figure 3: Analyzing communication for Case II

1. $i1_j \neq i2_j$, **or**,
2. $c1_j \neq c2_j$, **or**,
3. $d1_j \neq d2_j$ **and** either $d1_j$ or $d2_j$ is not a compile-time constant, **or**,
4. $i1_j = \phi, i2_j = \phi$ **and** $d1_j \neq d2_j$.

The first condition states that there is loop index permutation. In that case, a regular section move will be required. The second condition states that, along the j^{th} dimension, the elements of the arrays A and B are being accessed with different strides. Again, this case will require a regular section move. The third condition corresponds to the fact that there are non-constant offsets. If there are constant offsets, then only the overlap cells need to be filled in. For overlap cells, space needs to be allocated at compile-time, so the number of overlap cells must be known at the compile-time. If the offsets are not compile- time constants, then we use a regular section move to handle communication. This situation can also be handled by shifts into a temporary array [4]. The fourth condition says that along dimension j a loop variable does not appear in either the left hand side or the right hand side index and the loop invariant scalars are different. This represents a copy from one location to another, but because of the loop variables associated with other dimensions, will typically require a regular section of data to be moved. Since the distributed array descriptor is not available at compile-time, it cannot be determined at compile-time whether this data move will require any interprocessor communication. So we handle this kind of data move using the regular section move primitives we have already discussed.

The data access pattern requires filling in overlap cells, if the following condition holds:

1. A regular section move is not required **and**
2. $\exists j \in DD(A)$ s.t. $d1_j \neq d2_j$.

The second condition states that there is a difference in the offsets along some (distributed) dimension. Overlap cells must be filled along each dimension in which there is a difference in the offsets. In Figure 3 we show examples for the different possibilities within case II, for identically aligned two dimensional arrays A and B.

4.2.3 Case III

In this case, arrays A and B are aligned to the same template (T), but in different fashions. We consider only the cases when A and B are aligned to T in such a way that none of the dimensions of either A or B is replicated. In this case, the number of distributed dimensions of A and B would be identical, and will be equal to the number of distributed dimensions of T. Consider any distributed dimension j of A (i.e. $j \in DD(A)$). We use $AD(A, j)$ to denote the dimension of the template T along which the distributed dimension j of the array A is aligned. We use $map(j)$ to denote the dimension of the right hand side array B which is aligned to the same dimension of the template T as dimension j of the array A. Formally,

$$map(j) = k \iff \exists\, l \text{ s.t.} \quad AD(A, j) = l \wedge AD(B, k) = l \ .$$

Since each of the dimensions of A and B are distributed along exactly one dimension of the template T (as required by HPF), $map(j)$ is defined and is unique for each distributed dimension of A.

For the purpose of the discussion, we assume that the arrays A and B are aligned as follows:
!HPF$ ALIGN A $(k_1, \ldots, k_j, \ldots k_l)$ WITH T$(h1_1, \ldots, h1_{j'} : Ext_low1_j : Ext_high1_j, \ldots, h1_p)$
!HPF$ ALIGN B $(k_1, \ldots, k_j, \ldots k_n)$ WITH T$(h2_1, \ldots, h2_{j''} : Ext_low2_j : Ext_high2_j, \ldots, h2_p)$

where,

$$\begin{aligned} p &= |DD(A)| \\ j' &= AD(A, j) \\ j'' &= AD(B, j) \\ h1_{j'} &= a1_j * k_j + b1_j \\ h2_{j''} &= a2_j * k_j + b2_j \end{aligned}$$

In the above, dimension j of the array A is aligned with dimension j' of the template T. Similarly, dimension j of the array B is aligned with dimension j'' of the template T. Ext_low1_j and Ext_high1_j are, respectively, the number of external ghost cells at the beginning and end of dimension j of the Array A. Similarly, Ext_low2_j and Ext_high2_j are the number of external ghost cells at the beginning and end, respectively, of dimension j of Array B.

If we view the computation as accessing elements of the template T, then the effective offset for the left hand side array reference along dimension j of the array A is $d1_j * a1_j + b1_j - Ext\_low1_j$. Similarly, the effective offset for the right hand side array reference along dimension j of the array B is $d2_j * a2_j + b2_j - Ext\_low2_j$. The data access pattern in the *forall* loop will require a regular section move if the following condition holds:
$\exists j \in DD(A)$ s.t.

1. $i1_j \neq i2_{map(j)}$, **or**
2. $c1_j * a1_j \neq c2_{map(j)} * a2_{map(j)}$, **or**
3. $d1_j * a1_j + b1_j - Ext\_low1_j \neq d2_{map(j)} * a2_{map(j)} + b2_{map(j)} - Ext\_low2_{map(j)}$
   and either of these effective offsets is not a compile-time constant, **or**
4. $i1_j = \phi$, $i2_j = \phi$ **and** $b1_j - Ext\_low1_j \neq b2_{map(j)} - Ext\_low2_{map(j)}$.

As in Case II, the first condition states that there is loop index permutation. The second condition implies that there is a difference in the effective stride taken along any dimension. The third conditions says that the offsets may be distinct and one of them is not a compile-time constant. The fourth condition says that a rectilinear section of data needs to be moved along a certain dimension. Suppose that $i1_j = i_{k1}$ and

ALIGN A(i,j) WITH T(i,j)
ALIGN B(i,j) WITH T(2∗j,i+3:2:1)

L.H.S. Expression	R.H.S. Expression	Regular Section Move Required	Overlap Cell Fill Required
A(i,j)	B(i,j)	YES	NO
A(i,j)	B(j,i)	YES	NO
A(2*i,j)	B(j,i)	NO	YES
A(2*i,j)	B(j,i+3)	NO	YES
A(2*i,j)	B(j,i+1)	NO	NO

Figure 4: Analyzing communication for Case III

$i2_{map(j)} = i_{k2}$. If we view the loop iterations as accessing elements of the template T, $c1_j * st_{k1} * a1_j$ is the effective stride of the subscript on the left along dimension $AD(A, j)$ and similarly $c2_{map(j)} * st_{k2} * a2_{map(j)}$ is the effective stride of the subscript on the right hand side along dimension $AD(B, map(j))$. By the definition of $map(j)$, $AD(A, j) = AD(B, map(j))$. If $i1_j$ and $i2_{map(j)}$ are identical, then $k1 = k2$. So, the required condition for the effective stride for left and right side to be identical is $c1_j * a1_j = c2_{map(j)} * a2_{map(j)}$.

The data access pattern will require filling in overlap cells if the following condition holds:

1. A regular section move is not required **and**
2. $\exists j \in DD(A)$ s.t.
   $d1_j * a1_j + b1_j - Ext\_low1_j \neq d2_{map(j)} * a2_{map(j)} + b2_{map(j)} - Ext\_low2_{map(j)}$

The first condition says that we have not already decided that a regular section move was required. The second condition says that the effective offsets are not identical. In Figure 4, we give several examples showing the results of the analysis for Case III.

4.3 Generating calls to the runtime library

Once the nature of the communication required has been identified, the compiler must insert the appropriate calls to the runtime primitives. We first discuss how the calls are made to the routines for filling in ghost cells. We discuss this in the context of Case III from the previous section, since Case II is really a special case of Case III.

We identify each distributed dimension j of the array A for which

$$d1_j * a1_j + b1_j - Ext\_low1_j \neq d2_{map(j)} * a2_{map(j)} + b2_{map(j)} - Ext\_low2_{map(j)}.$$

One call to the schedule building primitive *Overlap_Cell_Fill_Sched* and one to the data moving primitive *Data_Move* is inserted for each such dimension. Since all computations are distributed using the owner computes rule, overlap cells are filled in for the right hand side array B. For the *Overlap_Cell_Fill_Sched* call, the dimension of the move is $map(j)$ and the number of overlap cells to be filled in is $d2_{map(j)} * a2_{map(j)} + b2_{map(j)} - Ext\_low2_{map(j)} - d1_j * a1_j + b1_j - Ext\_low1_j$.

The schedule building primitive is called with the Distributed Array Descriptor (DAD) of the array as a parameter. The actual array storage location need not be specified. A call to *Data_Move* is then made which uses the previously built schedule to copy the data.

We now discuss how calls to the primitive for moving regular sections are inserted. If there is more than one array on the right hand side, then the analysis described in Section 4.2 is done for each such array. For each of the right hand side arrays which requires a regular section move, a temporary array is declared and a regular section move is done from the right hand side array into the temporary array. If there is only one array on the right side (i.e. the *forall* loop represents only a copy and does not have any computation), then the regular section move is performed directly from the right hand side array to the left hand side array.

The parameters of *Regular_section_move_sched* are assigned as follows. In the *forall* loop, i_j is the j^{th} loop variable. The total number of loop variables is m. For each of the loop variables, we identify the dimensions corresponding to the subscripts of A and B where they appear. Srcdim(j) and Destdim(j) denote the dimensions of the source array B and the destination array A (or a temporary array) that correspond to the j^{th} dimension of the regular section being moved. Note that Srcdim(j) is not necessarily j since the *forall* loop allows arbitrary permutation among dimensions. If $i1_{k1} \equiv i_j$ and $i2_{k2} \equiv i_j$, then SrcDim(j) is assigned $k1$ and DestDim(j) is assigned $k2$. The remaining elements of Srcdims and Destdims are assigned the remaining dimensions of A and B whose subscripts are loop invariant scalars (the exact ordering is not important).

SrcLos(k) and SrcHis(k) denote the start and end points, respectively, along each dimension of the source. If $i2_k = \phi$, then, SrcLos(k) = SrcHis(k) = $d2_k$. Otherwise, if $i2_k \equiv i_j$, for some j, then, SrcLos(k) = $c2_k * (lo_j) + d2_k$ and similarly, SrcHis(k) = $c2_k * (hi_j) + d2_k$. For the destination, the low and high indexes are computed in the same manner.

SrcStr(k) denotes the stride of the move along each dimension of the source. If $i2_k = \phi$, then, SrcStr(k) doesn't matter. Otherwise, if $i2_k \equiv i_j$, for some j, then, SrcStr(k) = $c2_k * st_j$. For the destination, the strides are computed in the same manner.

4.4 Distributing loop iterations

Once the calls have been inserted for communicating the required array elements, the loop iterations must be distributed among the processors. As we stated earlier, this is done using the owner computes rule. Since the distributed array descriptors are built at runtime, it is not possible to compute the local loop bounds on each processor at compile-time.

Consider any loop variable i_j, Let $i2_{k1} \equiv i_j$. The loop accesses elements ranging from $c1_{k1} * lo_j + d1_{k1}$ to $c1_{k1} * hi_j + d1_{k1}$. We partition the loop based upon the portion of the distributed arrays that are owned by a given processor. This is done by inserting runtime calls to the the library primitives *Local_Lower_Bound* and *Local_Upper_Bound*. Note that for arrays which are not in canonical form (i.e. where $c1_j \neq 1$ or $d1_j \neq 0$ for some j), we can still partition the loop based upon the owners compute rule. Consequently, we never need to scatter any data after the loop has been executed.

In Figure 5 we show an example of how the calls to primitives for filling in overlap cells are inserted by the compiler. In Figure 6, we show how the compiler inserts calls to the primitives for moving regular sections. In both examples, the transformed code containing the calls to the runtime library will run as SPMD code on each processor of the distributed memory parallel machine. Note that in the compiler generated code, schedule building primitives will be called every time any *forall* loop requiring communication is executed. The hand coded version can build a schedule once and reuse it in subsequent iterations. Similarly, in the compiler generated code, runtime calls to the loop bound adjustment primitives will be made each time a loop is executed. The hand coded version can reuse the adjusted bounds over the multiple time steps, and also

```
C       ORIGINAL F90D CODE
C       Arrays A, B and C are distributed identically
          FORALL (i = 1:100,j = 1:100) A(i,j) = B(i+1,j) + C(i,j)

C       TRANSFORMED CODE
        Dim   =     1
        No_Of_Cells = 1
        sched  =    Overlap_Cell_Fill_Sched(DAD,Dim,No_Of_Cells)
C       DAD is distributed array descriptor for A, B and C
C       i is dimension 1, j is dimension 2
        Call   Data_Move(B,sched,B)
        L1     =    Local_Lower_Bound(DAD,1)
        L2     =    Local_Lower_Bound(DAD,2)
        H1     =    Local_Upper_Bound(DAD,1)
        H2     =    Local_Upper_Bound(DAD,2)
        do 10 i = L1,H1
        do 10 j = L2, H2
 10     A(i,j)  =    B(i+1,j) + C(i,j)
```

Figure 5: Overlap cell fill and loop bounds adjustment example

for multiple loops that have the same loop bounds. These two factors may cause compiler generated code to perform worse than hand parallelized code.

5 Experimental Results

In this section we present experimental results to demonstrate the efficacy of our approach. We are interested in two different factors, performance of the library primitives and the effectiveness of the compiler. We study the first factor by measuring the runtime overhead incurred in using our library primitives as compared the bare cost of communication associated with the best possible hand parallelized codes. We study the latter factor by comparing the performance of compiler parallelized codes with the codes parallelized by manually inserting calls to the library functions. We also study the effect of data distribution on the performance of these codes.

We have experimented with two major codes: a template from a multiblock Navier Stokes' solver and a semi-coarsening multigrid code. We have parallelized a template from a multiblock computation fluid dynamics application that solves the thin-layer Navier-Stokes equations over a 3D surface (multiblock TLNS3D), using our prototype Fortran 90D compiler. The multiblock TLNS3D code we are working with was developed by Vatsa *et al.* [40] at NASA Langley Research Center, and consists of nearly 18,000 lines of Fortran 77 code. The template, which was designed to include portions of the entire code that are representative of the major computation and communication patterns of the original code, consists of nearly 2,000 lines of F77 code. We have also worked with a semi-coarsening multigrid code [35]. This has nearly 2,500 lines of F77 code. In all the experiments described in this section, performance data is presented starting from the minimum number of processors which provided sufficient memory for executing the program up to 32 processors.

```
C     ORIGINAL F90D CODE
C     Arrays A, B are distributed identically
        forall (i = 1:100:2,j = 1:50) A(i,j) = B(2*j,i)

C     TRANSFORMED CODE
      NumSrcDim =  2          NumDestDim =  2
      SrcDim(1)  =  2         DestDim(1)  =  1
      SrcDim(2)  =  1         DestDim(2)  =  2
      SrcLos(1)  =  2         DestLos(1)  =  1
      SrcLos(2)  =  1         DestLos(2)  =  1
      SrcHis(1)  =  100       DestHis(1)  =  100
      SrcHis(2)  =  100       DestHis(2)  =  50
      SrcStr(1)  =  2         DestStr(1)  =  2
      SrcStr(2)  =  2         DestStr(2)  =  1
      Sched      =  Regular_Section_Move_Sched(DAD,DAD,NumSrcDim,NumDestDim,
                    SrcDim, SrcLos, SrcHis, SrcStr,
                    DestDim, DestLos, DestHis, DestStr)
      Call Data_Move(B,Sched,A)
```

Figure 6: Regular section move example

5.1 Overhead of Primitives

We are interested in evaluating the performance of a code parallelized using our primitives as compared the performance of the "best hand-parallelized" code. By hand parallelized code, here we mean parallelized by inserting calls to the communication routines provided by the distributed memory machine and not using our library routines. Note that, to the best of our knowledge, no complete block structured code has yet been hand-parallelized on a distributed memory parallel machines. The reason is that, for an application programmer parallelizing such a code with hand, it is very difficult to analyze the exact communication required in these codes and then be able to use the communication routines available on the parallel machine to handle it efficiently. The use of library primitives involves runtime overheads because of generating schedules, overhead of copying data to be communicated into buffer at source processors, and similarly the overhead of copying the received data into appropriate memory locations at the destination processors. The possible advantage (in terms of efficiency) of the library primitives is that, for each invocation of a data-move, each processor sends at most one message to each other processor. It may, in general, be very difficult for an application programmer, parallelizing the code by hand, to do such message aggregation. However, the best performance that an application programmer can ever achieve will only have the cost of actual communication and computation, assuming that messages have been aggregated to reduce the effect of communication latencies. We will study the overheads incurred in a code parallelized using our primitives as compared to the best possible performance of hand parallelized code, which incurs the minimum communication costs (assuming maximum message aggregation).

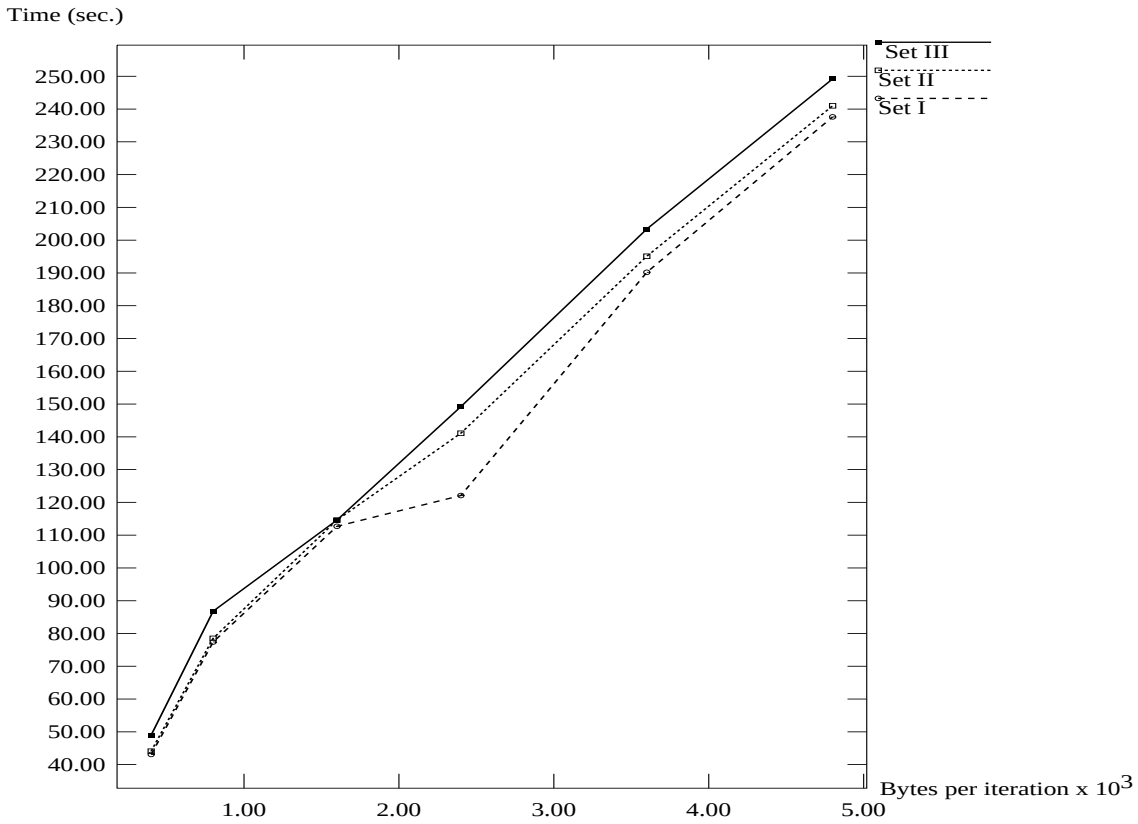
We consider a simple code executed on 2 processors in which a regular section move involves moving data from processor 0 to processor 1. We vary the number of bytes involved in the regular section moves and measure three sets of timings: the time required just for communication, the time required for communication when library primitives are used (excluding the cost of schedule building) and the total time required when the library primitives are used (including time for schedule building). The first set of timings represents the best performance that hand parallelization can achieve if all the data elements to be communicated are laid out contiguously. If the data elements to be communicated are not contiguous, then the application programmer will need to do copying to aggregate message. The second set of timings represents this case. The third set of timings represent the performance with the use of library primitives. The timings presented are for 100 iterations of the regular section move.

The performance results on an iPSC/860 are presented in Figure 7. The results show that the cost of copying (difference between Set I and Set II) is typically a small fraction (less than 5%) of the cost of communication for most of the cases. Also, if the schedule built is used over a large number of iterations, the cost of building the schedule is also a small fraction of the cost of communication. We have performed similar experiments on Thinking Machines' CM-5. Our experiments have shown that the runtime overheads associated with the use of primitives are again a small fraction of the bare cost of communication.

5.2 Multiblock Code

We have parallelized a multiblock template using our compiler. We hand parallelized this template by manually inserting calls to the multiblock Parti routines. (Note that this is different from the hand parallelization we talked about in the previous subsection.) We converted the F77 (sequential) code to F90D manually, by rewriting the the major computational parts of the code using *forall* loops and F90 array expressions, also adding the required data distribution directives. We then parallelized the code by running it through the F90D compiler. We also created a hand parallelized F90 version of the template in which all computations are done with single statement *forall* loops, but the calls to the runtime primitives are inserted manually.

We now compare the relative performances of compiler parallelized F90 code, hand parallelized F90 code



Set I : Communication (Max. message aggregation)
Set II : Communication and copying
Set III : Communication, copying and schedule building

Figure 7: Performance of Primitives on iPSC/860 (100 iterations)

ONE BLOCK: 49 X 9 X 9 Mesh (50 Iterations)

Number of Processors	Compiler Parallelized	Hand Parallelized F90	Hand Parallelized F77
4	6.99	6.88	6.20
8	4.17	4.06	4.00
16	2.47	2.35	2.28
32	1.55	1.45	1.41

Figure 8: Performance comparison for small mesh, one block (in seconds) on iPSC/860

TWO BLOCKS: 49 X 9 X 9 Each (50 Iterations)

Number of Processors	Compiler Parallelized	Hand Parallelized F90	Hand Parallelized F77
8	7.49	6.69	6.17
16	4.64	4.07	4.03
32	2.88	2.32	2.30

Figure 9: Performance comparison for larger mesh, two blocks (in seconds) on iPSC/860

and hand parallelized F77 code, varying the mesh size and number of blocks for the application, and also varying the number of processors used on an Intel iPSC/860. we used the minimum number of processors In Figure 8, we present the performance results on a $49 \times 9 \times 9$ mesh (with one block), comparing the performance of the three versions from 4 to 32 processors. In Figure 9, we present the performance results on a $49 \times 17 \times 9$ mesh (split into two blocks), comparing the performance of the three versions from 8 to 32 processors. The template is communication intensive and therefore the absolute speedups are not very high in either of the versions. The compiler parallelized F90 code performs within around 20% of the hand parallelized F77 code. The hand parallelized F90 code performs worse than the hand parallelized F77 code. This is because, in the F90 version, all computation is done through single statement *forall* loops that result in the creation of (large) temporary arrays. Such use of temporary storage, and the fact that no loop fusion between parallel loops is done by the compiler, increases the number of cache misses on each processor. However, the difference in performance between the F90 and F77 hand parallelized versions decreases as the number of processors increases. This is because as the number of processor increases, less memory is required on each processor, so the effect on cache utilization is less significant. The difference in performance of the hand parallelized F90 and the compiler parallelized code comes from two major factors. First, in the compiler generated version, the runtime calls for computing new loop bounds are made in each loop iteration, as compared to only once for the hand parallelized version. Second, as the template is run over a large number of time steps, the compiler generated version makes repeated calls to the runtime library to build communication schedules, whereas in the hand parallelized version the calls are lifted out of the time step loop. To reduce the additional cost due to this second factor, our runtime library library saves schedules. When a call is made for generating the schedule, the library searches a hash table to check if any schedule with exactly the same parameters is present. If so, the saved schedule is returned. This technique still has this overhead as compared to a hand parallelized version. To study the exact costs of each of these factors, we present a more detailed experiment in Section 5.4.

5.3 Multigrid Code

We have also experimented with a semi-coarsening multigrid code developed by Rosendale and Overman [35]. This has nearly 2,500 lines of F77 code. We discussed the semi-coarsening multigrid technique and the mapping policy used in parallelizing such an application earlier in Section 2.

We rewrote this code using forall loops and including the distribution directives and then parallelized it using our compiler. This code had also been parallelized by inserting the calls to the library routines manually [35]. In Figure 10, we show the performance comparison of these two parallel versions run on Intel iPSC/860. The results are for a $32x32x32$ grid, using a coarsening factor of 4 along each dimension. The code

No. of Proc.	Compiler: 1^{st} iteration	Compiler: Per Subsequent Iteration	By hand: 1^{st} iteration	By hand: Per Subsequent Iteration
8	4.80	2.29	4.60	2.14
16	3.84	1.38	3.41	1.35
32	3.03	.95	2.48	.88

Figure 10: Semi-Coarsening multigrid performance (in seconds) on iPSC/860

uses 8 different grids at four different levels. We did not create a separate hand parallelized F90 version since most of the subroutines in this code are fairly small and therefore rewriting it in F90 using forall loops did not involve introducing large temporary arrays. Consequently, we did not expect to see any notable difference in the performance of hand parallelized F90 and F77 versions.

The results of the performance of compiler parallelized and hand parallelized multigrid code are presented in Figure 10. The results show that the compiler parallelized code performs within 10% of the hand parallelized code in this case. Again, as this code was very communication intensive, the absolute speedup is not very high for either version.

5.4 Compiler Optimizations

In Figure 11, we study the effect of the compiler optimizations. Version I is a compiler parallelized version in which the library does not save any schedules. This version performs badly because of the high cost of rebuilding the schedules for every iteration. Version II is the compiler parallelized version in which the library saves schedules. This results in a major gain in performance. We now discuss some optimizations which are not implemented in the current compiler. We studied the effect of these optimizations by modifying the compiler generated code by hand. Version III represents the case where the compiler performs sophisticated interprocedural analysis to reuse the schedules during successive time steps. Version III performs better than version II, in which the schedules are reused within the library, but the difference is not large.

In the compiler parallelized version, runtime calls are made to the functions for adjusting loop bounds for each forall loop on each time step. The hand parallelized version can store the loop bounds computed during the first time step, for subsequent reuse. Additionally, a procedure may contain several loops involving the same array on the left hand side that have the same loop bounds. Our compiler generates separate runtime calls for adjusting loop bounds for each such loop. Such optimizations will be implemented in a future version of the compiler.

In Figure 11, the difference between version III and the hand parallelized F90 version shows the extra cost of generating loop bounds at runtime for each forall loop during each time step. The results show that generating loop bounds at runtime is the major factor in the performance difference between the compiler parallelized version and the hand parallelized versions. In version IV, we show the results of an unimplemented optimization in which the compiler is able to identify the loops with the same left hand side array and same loop bounds within a subroutine. Then the compiler needs to generate calls to the loop bound adjustment functions only once for each such set of loops. This optimization also provides an improvement over version III.

TWO BLOCKS : 25 X 9 X 9 Each (50 Iterations)

No. of Proc.	Compiler Version I	Compiler Version II	Compiler Version III	Compiler Version IV	Hand F90
4	13.45	7.63	7.41	7.33	6.79
8	15.51	4.78	4.58	4.54	4.19
16	11.72	2.85	2.71	2.62	2.39
32	8.01	1.85	1.79	1.66	1.47

Version I : Runtime Library does not save schedules
Version II : Runtime Library saves schedules
Version III : Schedules reuse implemented by hand
Version IV : Loop bounds reused within a procedure

Figure 11: Effects of various optimizations (in seconds) on iPSC/860

5.5 Effect of Data Distributions

As we discussed earlier, one of the features of our runtime library is the ability to map arrays (or templates) to subsets of the processor space. In the current definition of HPF (and hence in HPF compilers), this is not possible. In block structured codes, this feature allows us to keep the communication overheads low while maintaining the load balance. To study the benefit of this feature, we experimented with the multigrid template described above for two block case. We ran the parallelized code, once distributing both the blocks over the entire processor space and then distributing each block over disjoint processor spaces. The results on Intel iPSC/860, shown in the Figure 12, show that the latter scheme improves the performance by nearly 10 to 25%. Since there is no difference in the net computation performed at each Processor in either of the the two cases, this difference comes because of the increased amount of communication required when each block is distributed across the entire processor space. Mapping a block over a large number of processors increases communication arising from near neighbor interactions during the regular computation within blocks. Note that a 10 to 25% degradation in performance occurs when there are only two blocks. We expect that with a larger number of blocks, the difference in the performance would be much more severe.

6 Conclusions

To reliably and portably program distributed memory parallel machines , it is important to have both a machine independent language and runtime support for optimizing communication. High Performance Fortran and its variants have emerged as the most likely candidates for machine independent parallel programming on distributed memory machines. One class of scientific and engineering applications involves structured grids or meshes. These meshes may be nested (as in multigrid codes) or may be irregularly coupled (called multiblock codes or Irregularly Coupled Regular Meshes). Multiblock and multigrid codes form a significant part of scientific and engineering applications.

In this paper we have addressed the problem of runtime, compiler and programming language support

TWO BLOCK: 49 X 17 X 9 Mesh (50 Iterations)

Number of Processors	Blocks Mapped Entire Proc. Space	Blocks Mapped Disjoint Proc. Spaces
4	8.99	7.59
8	5.14	4.74
16	3.24	2.83
32	2.41	1.87

Figure 12: Effect of Data Distribution on iPSC/860

for parallelizing this important class of applications on distributed memory machines. We have designed and implemented a set of runtime primitives for parallelizing these applications in an efficient, convenient and machine independent manner. The runtime primitives give ability to specify data distributions, perform communication and distribute loops based on data distributions specified at runtime. One of the communication primitives in our library is the regular section move, which can copy a rectilinear part of a distributed array onto a rectilinear part of another distributed array, potentially involving index permutations, change of strides and change in offsets. We have presented runtime analysis which can implement this communication primitive efficiently.

For making the task of application programmers easy, it is important to have compiler support. In this paper, we have presented techniques that can be used by compilers for HPF-style programming languages to automatically generate calls to the runtime primitives. We have presented the method by which the compiler can analyze the data access patterns associated with parallel loops and therefore identify communication patterns which can be efficiently handled using the communication primitives that the multiblock Parti library supports. We have also presented compiler transformations that the compiler performs for automatically generating calls to the runtime primitives.

We have implemented the compiler analysis method in the Fortran 90D compiler being developed at Syracuse University. We consider this work to be a part of an integrated effort toward developing a powerful runtime support system for a F90D compiler. We have experimented with a template from a 3D multiblock Navier-Stokes solver and a multigrid code.

For demonstrating the efficacy of our approach, we examined two separate factors: performance of the runtime primitives and performance of compiler parallelized code as compared to the code parallelized by inserting calls to the runtime primitives by hand. We examined the additional cost of using library primitives as compared to the minimum cost of communication. Performance results show that the additional cost in using the library primitives (schedule building and data copying) is a small fraction of the minimum cost of communication. One of the features of our library which is not supported in current version of HPF (and consequently in HPF compilers) is the ability to map arrays or blocks over part of the processor space. We presented results with TLNS3D template to show the improvement in performance achieved because of this feature. We compared the performance of compiler parallelized code with the performance of hand parallelized F90 and F77 codes, and have shown that the compiler parallelized code performs within 20% of hand parallelized F77 code. The optimization of having the runtime library save and reuse communication schedules allows the compiler parallelized code to perform almost as well as hand parallelized code. We have also experimented with other optimizations. The optimization of reusing computed loop bounds within a

subroutine improves the performance of the compiler parallelized code and brings it within 10% of the hand parallelized version.

To the best of our knowledge, we are not aware of any real block structured codes which have been parallelized on any distributed memory machine. The reason is that, for an application programmer parallelizing such an application by hand, it is very difficult to analyze the exact communication required and then to be able to use the communication routines provided by the machine to communicate efficiently. Our runtime and compiler support can be used to parallelize such applications conveniently. Our experimental results have shown that the code parallelized by using the compiler will have only a small overhead as compared to the best hand parallelized code (i.e. parallelized by invoking system's communication primitives by hand).

While the design of our runtime system was motivated by multiblock and multigrid applications, our runtime primitives can be used in many cases for regular codes as well. We therefore, believe that our runtime support and compiler techniques can be used by compilers of HPF-style parallel programming languages in general.

Acknowledgements

We gratefully acknowledge our collaborators, Geoffrey Fox, Alok Choudhary, Sanjay Ranka, Tomasz Haupt, and Zeki Bozkus for many enlightening discussions and for allowing us to integrate our runtime support into their emerging Fortran 90D compiler. The detailed discussions we had with Sanjay Ranka, Alok Choudhary and Zeki Bozkus during their visits to Maryland were extremely productive. We are also grateful to V. Vatsa and M. Senetrik at NASA Langley for giving us access to the multiblock TLNS3D application code. We will also like to thank John van Rosendale at ICASE and Andrea Overman at NASA Langley for making their sequential and hand parallelized multigrid code available to us. We also thank Jim Humphries for creating a portable version of the runtime library.

References

[1] Christopher A. Atwood. Selected computations of transonic cavity flows. In *Proceedings of the 1993 ASME Fluids Engineering Conference, Forum on Computational Aero- and Hydro-Acoustics*, June 1993.

[2] M.J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:67–84, 1989.

[3] M.J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. Submitted to the Journal of Parallel and Distributed Computing, March 1993.

[5] W. Briggs. *A Multigird Tutorial*. SIAM, 1987.

[6] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[7] Siddhartha Chatterjee, John R. Gilbert, Fred J.E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 149–158, May 1993. ACM SIGPLAN Notices, Vol. 28, No. 7.

[8] Kalpana Chawla and William R. Van Dalsem. Numerical simulation of a powered-lift landing. In *Proceedings of the 72nd Fluid Dynamics Panel Meeting and Symposium on Computational and Experimental Assessment of Jets in Cross Flow, Winchester, UK*. AGARD, April 1993.

[9] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3(1-4):43–52, 1992. Papers presented at the Symposium on High-Performance Computing for Flight Vehicles, December 1992.

[10] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pages 185–220. Elsevier, 1992.

[11] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. Technical Report CS-TR-3076 and UMIACS-TR-93-42, University of Maryland, Department of Computer Science and UMIACS, May 1993. To appear in LCPC '93.

[12] D. Loveman (Ed.). Draft High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

[13] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[14] Survey of principal investigators of grand challenge applications: Workshop on grand challenge applications and software technology, May 1993.

[15] Al Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[16] Michael Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[17] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed memory machines. In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[18] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[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, July 1991.

[20] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the Sixth International Conference on Supercomputing*. ACM Press, July 1992.

[21] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings Supercomputing '91*, pages 86–100. IEEE Computer Society Press, November 1991.

[22] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[23] J.R.G.Townshend, C.O.Justice, W. Li, C.Gurney, and J.McManus. Global land cover classification by remote sensing:present capabilities and future possibilities. *Remote Sensing of Environment*, 35:243–256, 1991.

[24] Scott R. Kohn and Scott B. Baden. An implementation of the LPAR parallel programming model for scientific computations. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 759–766. SIAM, March 1993.

[25] George Lake. Personal communication.

[26] Max Lemke and Daniel Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. Technical Report 611, GMD, February 1992.

[27] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[28] R. Mathur, L.K. Peters, and R.D. Saylor. Sub-grid representation of emission source clusters in regional air quality modeling. *Atmospheric Environment*, 26A:3219–3238, 1992.

[29] S. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*. SIAM, 1989.

[30] S. McCormick. *Multilevel Projection Methods for Partial Differential Equations*. SIAM, 1992.

[31] R. Meakin. Moving body overset grid methods for complete aircraft tiltrotor simulations, AIAA-93-3350. In *Proceedings of the 11th AIAA Computational Fluid Dynamics Conference*, July 1993.

[32] Odman M.T. and A.G. Russell. A multiscale finite element pollutant transport scheme for urban and regional modeling. *Atmospheric Environment*, 25A:2385–2398, 1991.

[33] Richard Muntz. Personal communication.

[34] Naomi H. Naik and John Van Rosendale. The improved robustness of multigrid elliptic solvers based on multiple semicoarsened grids. *SIAM Journal of Numerical Analysis*, 30(1):215–229, February 1993.

[35] Andrea Overman and John Van Rosendale. Mapping robust parallel multigrid algorithms to scalable memory architectures. To appear in Proceedings of 1993 Copper Mountain Conference on Multigrid Methods, April 1993.

[36] J.J. Quirk. *An Adaptive Grid Algorithm for Computational Shock Hydrodynamics*. PhD thesis, Cranfield Institute of Technology, January 1991.

[37] James M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[38] J.M. Stone and M.L. Norman. Zeus-2d: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions: I. the hydrodynamic algorithms and tests. *The Astrophysical Journal Supplements*, 80(753), 1992.

[39] Alan Sussman and Joel Saltz. A manual for the multiblock PARTI runtime primitives. Technical Report CS-TR-3070 and UMIACS-TR-93-36, University of Maryland, Department of Computer Science and UMIACS, May 1993.

[40] V.N. Vatsa, M.D. Sanetrik, and E.B. Parlette. Development of a flexible and efficient multigrid-based multiblock flow solver; AIAA-93-0677. In *Proceedings of the 31st Aerospace Sciences Meeting and Exhibit*, January 1993.

[41] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran — a language specification, version 1.1. Interim Report 21, ICASE, NASA Langley Research Center, March 1992.