Syracuse University
SURFACE

Northeast Parallel Architecture Center | College of Engineering and Computer Science

1993

A Compilation Approach for Fortran 90D/HPF Compilers on Distributed Memory MIMD Computers

Zeki Bozkus
Syracuse University, Northeast Parallel Architectures Center, zbozkus@npac.syr.edu

Alok Choudhary
Syracuse University, Northeast Parallel Architectures Center

Geoffrey C. Fox
Syracuse University, Northeast Parallel Architectures Center

Tomasz Haupt
Syracuse University, Northeast Parallel Architectures Center, haupt@npac.syr.edu

Follow this and additional works at: http://surface.syr.edu/npac

Part of the Computer Sciences Commons

Recommended Citation
Bozkus, Zeki; Choudhary, Alok; Fox, Geoffrey C.; and Haupt, Tomasz, "A Compilation Approach for Fortran 90D/HPF Compilers on Distributed Memory MIMD Computers" (1993). *Northeast Parallel Architecture Center.* Paper 16.
http://surface.syr.edu/npac/16

This Working Paper is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Northeast Parallel Architecture Center by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

A Compilation Approach for Fortran 90D/HPF Compilers on Distributed Memory MIMD Computers*

Zeki Bozkus, Alok Choudhary†, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka
Northeast Parallel Architectures Center
3-201, Center for Science and Technology
Syracuse University
Syracuse, NY 13244-4100
{zbozkus, choudhar, gcf, haupt, ranka}@npac.syr.edu

May 3, 1993

Abstract

This paper describes a compilation approach for a Fortran 90D/HPF compiler, a source-to-source parallel compiler for distributed memory systems. Different from Fortran 77 parallelizing compilers, a Fortran90D/HPF compiler does not **parallelize** sequential constructs. Only parallelism expressed by Fortran 90D/HPF parallel constructs is exploited. The methodoly of parallelizing Fortran programs such as computation partitioning, communication detection and generation, and the run-time support for the compiler are discussed. An example of Gaussian Elimination is used to illustrate the compilation techniques with performance results.

*This work was supported in part by NSF under CCR-9110812 (Center for Research on Parallel Computation) and DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

†Corresponding Author: Alok Choudhary, 121 Link Hall, ECE Dept. Syracuse University, Syracuse, NY, 13244

1 Introduction

Distributed memory multiprocessors are increasingly being used for providing high performance for scientific applications. Distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, though it is widely accepted that they are difficult to program given the current status of software technology. Currently, distributed memory machines are programmed using a node language and a message passing library. This process is tedious and error prone because the user must perform the task of data distribution and communication for non-local data access.

There has been significant research in developing parallelizing compilers. In this approach, the compiler takes a sequential program, e.g. a Fortran 77 program as input, applies a set of transformation rules, and produces a parallelized code for the target machine. However, a sequential language, such as Fortran 77, obscures the parallelism of a problem in sequential loops and other sequential constructs. This makes the potential parallelism of a program more difficult to detect by a parallelizing compiler. Therefore, compiling a sequential program into a parallel program is not a natural approach. An alternative approach is to use a programming language that can naturally represent an application without losing the application's original parallelism. Fortran 90 [1] (with some extensions) is such a language. The extensions may include the *forall* statement and compiler directives for data partitioning, such as decomposition, alignment, and distribution. Fortran 90 with these extensions is what we call "Fortran 90D", a Fortran 90 version of the Fortran D language [2]. We developed the Fortran D language with our colleagues at Rice University. There is an analogous version of Fortran 77 with compiler directives and other constructs, called Fortran 77D. Fortran D allows the user to advise the compiler on the allocation of data to processor memories. Recently,the High Performance Fortran Forum, an informal group of people from academia, industry and national labs, led by Ken Kennedy, developed a language called HPF (High Performance Fortran) [3] based on Fortran D. HPF essentially adds extensions to Fortran 90 similar to Fortran D directives. Hence, Fortran 90D and HPF are very similar except a few differences. For this reason, we call our compiler the Fortran 90D/HPF compiler.

From our point of view, Fortran90 is not only a language for SIMD computers [4, 5], but it is a natural language for specifying parallelism in a class of problems called *loosely synchronous* problems [6]. In Fortran 90D/HPF, parallelism is represented with parallel constructs, such as array

operations, *where* statements, *forall* statements, and intrinsic functions. This gives the programmer a powerful tool to express the data parallelism natural to a problem.

This paper presents the design of a prototype compiler for Fortran 90D/HPF. The compiler takes as input a program written in Fortran 90D/HPF. Its output is SPMD (Single Program Multiple Data) program with appropriate data and computation partitioning and communication calls for MIMD machines. Therefore, the user can still program using a data parallel language but is relieved of the responsibility to perform data distribution and communication.

The goals of this paper are to present the underlying design philosophy, various design choices and the reasons for making these choices, and to describe our experience with the implementation. That is, in contrast to many other compiler papers which present specific techniques to perform one or more functions, our goal is to describe the overall architecture of our compiler. We believe that the presented design will provide directions to the implementors of HPF compilers.

Tremendeous effort in the last decade has been devoted to the goal of running existing Fortran programs on new parallel machines. Restructuring compilers for Fortran 77 programs have been researched extensivelly for shared memory systems[7, 8]. The compilation techinique of Fortran 77 for distributed memory systems has been addressed by Callahan and Kennedy [9]. Currently, a Fortran 77D compiler is being developed at Rice [10, 11]. Hatcher and Quinn provide a working version of a C* compiler. This work converts C* - an extension of C that incorporates dfeatures of a data parallel SIMD programming model- into C plus messasge passing for MIMD distributed memory parallel computes[12]. The ADAPT system [13] compiles Fortran 90 for execution on MIMD distributed memory architectures. The ADAPTOR [14] is a tool that transform data parallel programs written in Fortran with array extension and layout directives to explicit message passing. Chen [15, 16] describes general compiler optimization techniques that reduce communication overhead for Fortran-90 implementation on massivelly parallel machines. Many techniques especially for unstructured communication of Fortran 90D/HPF compiler are adapted from Saltz *et al.* [17, 18, 19]. Gupta *et al.* [20, 21] use collective communication on automatic data partitioning on distributed memory machines. uperb [22] compiles a Fortran 77 program into a semantically equivalent parallel SUPRENUM multiprocessor. Koelbel and Mehrotra [23, 24] present a compilation method where a great deal of effort is put on run-time analysis for optimizing message passing in implementation of Kali.

Figure 1: **Sketch of Fortran 90D/HPF compiler.**

2 Compilation Overview

Our Fortran90D/HPF parallel compiler exploits only the parallelism expressed in the data parallel constructs. We do not attempt to *parallelize* other constructs, such as *do* loops and *while* loops, since they are used only as naturally sequential control constructs in this language. The foundation of our design lies in recognizing commonly occurring computation and communication patterns. These patterns are then replaced by calls to the optimized run-time support system routines. The run-time support system includes parallel intrinsic functions, data distribution functions, communication primitives and several other miscellaneous routines (shown in Figure 1). This approach represents a significant departure from traditional approaches where a compiler needs to perform in-depth dependency analyses to recognize parallelism, and embed all the synchronization and low-level communication functions inside the generated code.

Given a syntactically correct Fortran90D/HPF program, the first step of the compilation is to generate a parse tree. The front-end to parse Fortran 90 for the compiler was obtained from ParaSoft

Corporation. In this module, our compiler also transforms each array assignment statement and *where* statement into equivalent *forall* statement with no loss of information [25]. In this way, the subsequent steps need only deal with *forall* statements.

The partitioning module processes the data distribution directives; namely, decomposition, distribute and align. Using these directives, it partitions data and computation among processors.

After partitioning, the parallel constructs in the node program are sequentialized since they would be executed on a single processor. This is performed by the sequentialization module. Array operations and *forall* statements in the original program are transferred into loops or nested loops. The communication module detects communication requirements and inserts appropriate communication primitives.

Finally, the code generator produces *loosely synchronous* [6] SPMD code. The generated code is structured as alternating phases of local computation and global communication. Local computations consist of operations by each processor on the data in its own memory. Global communication includes any transfer of data among processors, possibly with arithmetic or logical computation on the data as it is transferred (e.g. reduction functions). In such a model, processes do not need to synchronize during local computation. But, if two or more nodes interact, they are implicitly synchronized by global communication.

3 Data Partitioning

Distributed memory systems solve the memory bottleneck of vector supercomputers by having separate memory for each processor. However, distributed memory systems demand high locality for good performance. Therefore, the distribution of data across processors is of critical importance to the performance of a parallel program in a distributed memory system.

Fortran D provides users with explicit control over data partitioning with both data *alignment* and *distribution* specifications. We briefly overview directives of Fortran D relevant to this paper. The complete language is described elsewhere [2]. The DECOMPOSITION directive is used to declare the name, dimensionality, and the size of each problem domain. We call it "template" (the name "template" has been chosen to describe "DECOMPOSITION" in HPF [3]). The ALIGN directive specifies fine-grain parallelism, mapping each array element onto one or more elements of the template. This provides the minimal requirement for reducing data movement. The DIS-

TRIBUTE directive specifies coarse-grain parallelism, grouping template elements and mapping them to the finite resources of the machine. Each dimension of the template is distributed in either a block or cyclic fashion. The selected distribution can affect the ability of the compiler to minimize communication and load imbalance in the resulting program.

The DISTRIBUTE directive assigns an *attribute* to each dimension of the template. Each attribute describes the mapping of the data in that dimension of the template on the logical processor grid. The first version of the compiler supports the following types of distribution.

- **BLOCK** divides the template into contiguous chunks.
- **CYCLIC** specifies a round-robin division of the template.

The **BLOCK** attribute indicates that blocks of global indices are mapped to the same processor. The block size depends on the size of the template dimension, N, and the number of processors, P, on which that dimension is distributed (shown in the first column of Table 1).

Table 1: **Data distribution function(refer to Definition 1): N is the size of the global index space. P is the number of processors. N and P are known at compile time and $N \geq P$. I is the global index. i is the local index and p is the owner of that local index i.**

	Block-distribution	Cyclic-distribution
global to proc $I \rightarrow p$	$p = \frac{I*P}{N}$	$p = I \ mod \ P$
global to local $I \rightarrow i$	$i = I - \frac{p*N}{P}$	$i = \lfloor \frac{I}{P} \rfloor$
local to global $(p, i) \rightarrow I$	$I = i + \frac{p*N}{P}$	$I = iP + p$
cardinality	$\frac{N}{P}$	$\lfloor \frac{N+P-1-p}{P} \rfloor$

The **CYCLIC** attribute indicates that global indices of the template in the specified dimension should be assigned to the logical processors in a round-robin fashion. The last column of Table 1 shows the CYCLIC distribution functions. This also yields an optimal static load balance since the first $N \ mod \ P$ processors get $\lceil \frac{N}{P} \rceil$ elements; the rest get $\lfloor \frac{N}{P} \rfloor$ elements. In addition, these distribution functions are efficient and simple to compute. Although cyclic distribution functions

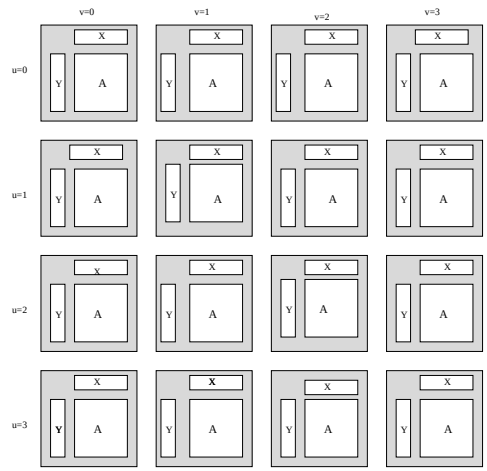

Figure 2: **Matrix-vector decomposition: each processor is assigned array section of A, X, and Y.**

provided a good static load balance, the locality is worse than that using block distribution because cyclic distributions scatter data.

The following example illustrates the Fortran D directives. Consider the data partitioning schema for matrix-vector multiplication proposed by Fox *et al.*[26] and shown in Figure 2. The matrix vector multiplication can be described as

$$y = Ax$$

where y and x are vectors of length M, and A is an $M \times M$ matrix. To create the distribution shown in the Figure 2, one can use the following directives in a Fortran 90D program.

```
C$   DECOMPOSITION    TEMPL(M,M)
C$   ALIGN A(I,J)  WITH TEMPL(I,J)
C$   ALIGN X(J)    WITH TEMPL(*,J)
C$   ALIGN Y(I)    WITH TEMPL(I,*)
C$   DISTRIBUTE   TEMPL(BLOCK,BLOCK)
```

If this program is mapped onto a 4x4 physical processor system, the Fortran 90D compiler will generate the distributions shown in Figure 2. Matrix A is distributed in both dimensions. Hence, a single processor owns a subset of matrix rows and columns. X is column-distributed and row-replicated. But Y is row-distributed and column-replicated.

4 Computation Partitioning

Once the data is distributed, there are several alternatives to assign computations to processing elements (PEs) for each instance of a *forall* statement. One of the most common methods is to use the *owner computes rule*. In the owner computes rule, the computation is assigned to the PE owning the *lhs* data element. This rule is simple to implement and performs well in a large number of cases. Most of the current implementations of parallelizing compilers uses the owner computes rule [22, 9]. However, it may not be possible to apply the owner computes rule for every case without extensive overhead. The following examples describe how our compiler performs computation partitioning.

Example 1 (canonical form) Consider the following statement, taken from the Jacobi relaxation program

```
 forall (i=1:N, j=1:N)
&   B(i,j) = 0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
```

In the above example, as in a large number of scientific computations, the *forall* statement can be written in the canonical form. In this form, the subscript value in the *lhs* is identical to the forall iteration variable. In such cases, the iterations can be easily distributed using the owner computes rule. Furthermore, it is also simpler to detect structured communication by using this form (This will be elaborated in Section 5.2.).

Figure 3 shows the possible data and iteration distributions for the $lhs_I = rhs_I$ assignment caused by iteration instance I. Cases 1 and 2 illustrate the order of communication and computation arising from the owner computes rule. Essentially, all the communications to fetch the off-processor data required to execute an iteration instance are performed before the computation is performed. The generated code will have the following communication and computation order.

```
Communications  ! some global communication primitives
Computation     ! local computation
```

Example 2 (non-canonical form) Consider the following statement, taken from an FFT program

```
 forall (i=1:incrm, j=1:nx/2)
&   x(i+j*incrm*2+incrm) = x(i+j*incrm*2) - term2(i+j*incrm*2+incrm)
```

The *lhs* array index is not in the canonical form. In this case, the compiler equally distributes the iteration space on the number of processors on which the *lhs* array is distributed. Hence, the

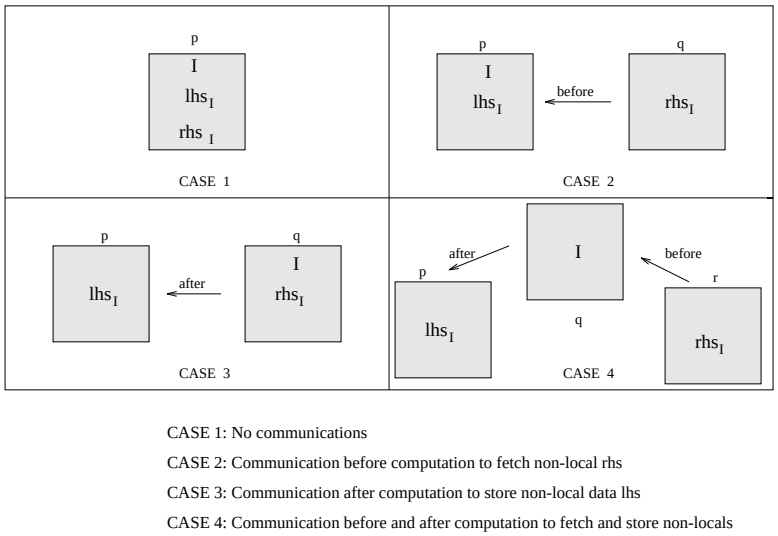

Figure 3: **I shows the processor on which the computation is performed.** lhs_I **and** rhs_I **show the processors on which the** lhs **and** rhs **of instance I reside.**

total number of iterations will still be the same as the number of lhs array elements being assigned. However, this type of forall statement will result in either Case 3 or Case 4 in Figure 2. The generated code will be in the following order.

```
Communications  ! some global communication primitives to read off-processor values
Computation     ! local computation
Communication   ! a communication primitive to write the calculated values to off-processors
```

For reasonably simple expressions, the compiler can transform such index expressions into the canonical form by performing some symbolic expression operations [27]. However, it may not always be possible to perform such transformations for complex expressions.

Example 3 (vector-valued index) Consider the statement

```
forall (i=1:N) A(U(i)) = B(V(i)) +C(i)
```

The iteration i causes an assignment to element $A(U(i))$, where $U(i)$ may only be known at run-time. Therefore, if iterations are statically assigned at compile time to various PEs, iteration i is likely to be assigned to a PE other than the one owning $A(U(i))$. This is also illustrated in cases 3 and 4 of Figure 3. In this case, our compiler distributes the computation i with respect to the owner of $A(i)$.

Having presented the computation partitioning alternatives for various reference patterns of arrays on the *lhs*, we now present a primitive to perform global to local transformations for loop bounds.

```
set_BOUND(llb,lub,lst,glb,gub,gst,DIST,dim) ! computes local lb, ub, st from global ones
```

The *set_BOUND* primitive takes a global computation range with global lower bound, upper bound and stride. It distributes this global range statically among the group of processors specified by the *dim* parameter on the logical processor dimension. The *DIST* parameter gives the distribution attribute such as *block* or *cyclic*. The *set_BOUND* primitive computes and returns the local computation range in local lower bound, local upper bound and local stride for each processor. The algorithm to implement this primitive can be found in [25].

The other functionality of the *set_BOUND* primitive is to mask inactive processors by returning appropriate local bounds. For example, such a case may arise when the global bounds do not specify the entire range of the *lhs* array. If the inputs for this primitive are compile-time constants, the compiler can calculate the local bounds at compile-time.

In summary, our computation and data distributions have two implications.

- The processor that is assigned an iteration is responsible for computing the *rhs* expression of the assignment statement.
- The processor that owns an array element (*lhs* or *rhs*) must communicate the value of that element to the processors performing the computation.

5 Communication

Our Fortran 90D/HPF compiler produces calls to collective communication routines instead of generating individual processor send and receive calls inside the compiled code. The idea of using collective communication routines came from researchers involved in developing scientific application programs [26]. There are three main reasons for using collective communication to support interprocessor communication in the Fortran 90D/HPF compiler.

1. *Improved performance of Fortran 90D/HPF programs.* To achieve good performance, interprocessor communication must be minimized. By developing a separate library of interpro-

cessor communication routines, each routine can be optimized. This is particularly important given that the routines will be used by many programs compiled through the compiler.

2. *Increased portability of the Fortran 90D/HPF compiler.* By separating the communication library from the basic compiler design, portability is enhanced because to port the compiler, only the machine specific low-level communication calls in the library need to be changed.

3. *Improved performance estimation of communication costs.* Our compiler takes the data distribution for the source arrays from the user as compiler directives. However, any future compiler will require a capability to perform automatic data distribution and alignments [28, 29, 30]. In any case, distributions of temporary arrays must be determined by the compiler. Such techniques usually require computing trade-offs between exploitable parallelism and the communication costs. The costs of collective communication routines can be determined more precisely, thereby enabling the compiler to generate better distributions.

In order to perform a collective communication on array elements, the communication primitive needs the following information 1-) send processors list, 2-) receive processors list, 3-) local index list of the source array and, 4-) local index list of the destination array.

There are two ways of determining the above information. 1) Using a pre-processing loop to compute the above values or, 2) based on the type of communication, the above information may be implicitly available, and therefore, not require pre-processing. We classify our communication primitives into *structured* and *unstructured* communication.

Our structured communication primitives are based on a logical grid configuration of the processors which is formed according to the shape of template and the number of available physical processors. Hence, they use grid-based communications such as shift along dimensions, broadcast along dimensions etc. The following summarizes some of the structured communication primitives implemented in our compiler.

- **transfer:** Single source to single destination message. This may happen that one column of grid processors communicate with another column of the grid processors.

- **multicast:** broadcast along a dimension of the logical grid.

- **overlap_shift:** shifting data into overlap areas in one or more grid dimensions. This is particularly useful when the shift amount is known at compile time. This primitive uses that fact to avoid intra processor copying of data and directly stores data in the overlap areas [31].

- **temporary_shift:** This is similar to overlap shift except that the data is shifted into a temporary array. This is useful when the shift amount is not a compile time constant. This shift may require intra-processor copying of data.

We have implemented two sets of unstructured communication primitives: 1) where the communicating processors can determine the send and receive lists based only on local information, and hence, only require pre-processing that involves local computations, [24] and 2) where to determine the send and receive lists pre-processing itself requires communication among the processors [19].

- **precomp_read:** This primitive is used to bring all non-local data to the place it is needed before the computation is performed.

- **postcomp_write:** This primitive is used to store remote data by sending it to the processors that own the data after the computation is performed. Note that these two primitives requires only local computation in the pre-prcessing loop.

- **gather:** This is similar to *precomp_read* except that pre-processing loop itself may require communication.

- **scatter:** This is similar to *postcomp_write* except that pre-processing loop itself may require communication.

The gather and scatter operations are powerful enough to provide the ability to read and write distributed arrays with vectorized communication facility. These two primitives are available in PARTI (Parallel Automatic Runtime Toolkit at ICASE) [32] designed to efficiently support irregular patterns of distributed array accesses. Fortran 90D/HPF compiler uses the PARTI to support these two powerful primitives.

The compiler must recognize the presence of collective communication patterns in the computations in order to generate the appropriate communication calls. Specifically, this involves a number of tests on the relationship among subscripts of various arrays in a forall statement. These tests should also include information about array alignments and distributions. We use pattern

matching techniques similar to those proposed by Chen [15]. Further, we extend the above tests to include unstructured communication. Table 2 shows the patterns of communication primitives used in our compiler. The detail of communication detection algorithm can be found in [25].

Table 2: **Communication primitives based on the relationship between *lhs* and *rhs* array subscript reference pattern for block distribution. (c: compile time constant, s, d: scalar, f: invertible function, V: an indirection array).**

Steps	(lhs,rhs)	Comm. primitives
1	(i, s)	multicast
2	$(i, i+c)$	overlap_shift
3	$(i, i-c)$	overlap_shift
4	$(i, i+s)$	temporary_shift
5	$(i, i-s)$	temporary_shift
6	(d, s)	transfer
7	(i, i)	no_communication
8	$(i, f(i))$	precomp_read
9	$(f(i), i)$	postcomp_write
10	$(i, V(i))$	gather
11	$(V(i), i)$	scatter
12	$(i, unknown)$	gather
13	$(unknown, i)$	gather

We would like give an example about the code generation about the unstructured communication at our compiler.(Communication generation for structured one can be found at Section 7). In distributed memory MIMD architectures, there is typically a non-trivial communication latency or startup cost. Hence, it is attractive to vectorize messages to reduce the number of startups. For unstructured communication, this optimization can be achieved by performing the entire pre-processing loop before communication so that the schedule routine can combine the messages to a maximum extent. The pre-processing loop is also called the "inspector" loop [32, 23].

Example 1 (precomp_read) Consider the statement

```
FORALL(I=1:N) A(I)=B(2*I+1)
```

The array B is marked as *precomp_read* since the distributed dimension subscript is written as $f(i) = 2 * i + 1$ which is invertible as $g(i) = (i - 1)/2$.

```
1       count=1
```

```
2      call set_BOUND(lb,ub,st,1,N,1) ! compute local lb, ub, and st
3      DO I=lb,ub,st
4         receive_list(count)=global_to_proc(f(i))
5         send_list(count)= global_to_proc(g(i))
6         local_list(count) = global_to_local(g(i))
7         count=count+1
8      END DO
9      isch = schedule1(receive_list, send_list, local_list, count)
10     call precomp_read(isch, tmp,B)
11     count=1
12     DO I=lb,ub,st
13        A(I) = tmp(count)
14        count= count+1
15     END DO
```

The pre-processing loop is given in lines 1-9. Note that this pre-processing loop executes concurrently in each processor. It fills out the *receive_list* as well as the *send_list* of processors. Each processor also fills the local indices of the array elements which are needed by that processor.

The schedule *isch* can also be used to carry out identical patterns of data exchanges on several different but identically distributed arrays or array sections. The same schedule can be reused to repeatedly carry out a particular pattern of data exchange on a single distributed array. In these cases, the cost of generating the schedules can be amortized by only executing it once. This analysis can be performed at compile time. Hence, if the compiler recognizes that the same schedule can be reused, it does not generate code for scheduling but it passes a pointer to the already existing schedule. Furthermore, the pre-processing computation can be moved up as much as possible by analyzing *definition-use chains* [33]. Reduction in communication overhead can be significant if by this analysis the scheduling code can be moved out of one or more nested loops.

In the above example, *local_list* (line 6) is used to store the index of one-dimensional array. However, in general, *local_list* will store indices from a multi-dimensional Fortran array by using the usual column-major subscript calculations to map the indices to a one-dimensional index.

The *precomp_read* primitive performs the actual communication using the schedule. Once the communication is performed, the data is ordered in a one dimensional array, and the computation (lines 12-15) uses this one dimensional array.

6 Run-time Support System

The Fortran 90D/HPF compiler relies on a very powerful run-time support system. The run-time support system consists of functions which can be called from the node programs of a distributed memory machine. Intrinsic functions support many of the basic data parallel operations in Fortran 90. They do not only provide a concise means of expressing operations on arrays, but also identify parallel computation patterns that may be difficult to detect automatically. Fortran 90 provides intrinsic functions for operations such as shift, reduction, transpose, and transpose, and matrix multiplication. The intrinsic functions that may induce communication can be divided into five categories as shown in Table 3.

Table 3: Fortran90D/HPF Intrinsic Functions

1. Structured communication	2. Reduction	3. Multicasting	4. Unstructured communication	5. Special routines
CSHIFT EOSHIFT	DOTPRODUCT ALL, ANY COUNT MAXVAL, MINVAL PRODUCT SUM MAXLOC, MINLOC	SPREAD	PACK UNPACK RESHAPE TRANSPOSE	MATMUL

The first category requires data to be transferred using with less overhead structured shift communications operations. The second category of intrinsic functions require computations based on local data followed by use of a reduction tree on the processors involved in the execution of the intrinsic function. The third category uses multiple broadcast trees to spread data. The fourth category is implemented using unstructured communication patterns. The fifth category is implemented using existing research on parallel matrix algorithms [26]. Some of the intrinsic functions can be further optimized for the underlying hardware architecture.

Table 4 presents a sample of performance numbers for a subset of the intrinsic functions on iPSC/860. A detailed performance study is presented in [34]. The times in the table include both the computation and communication times for each function. For most of the functions we were able to obtain almost linear speedups. In the case of TRANSPOSE function, going from one processor

to two or four actually results in increase in the time due to the communication requirements. However, for larger size multiprocessors the times decrease as expected.

Table 4: Performance of some Fortran 90D Intrinsic Functions (time is milliseconds).

Nproc	ALL (1K x 1K)	ANY (1K x 1K)	MAXVAL (1K x 1K)	PRODUCT (256K)	DOT PRODUCT (256K)	TRANSPOSE (512 x 512)
1	580.6	606.2	658.8	90.1	164.8	299.0
2	291.0	303.7	330.4	50.0	83.0	575.0
4	146.2	152.6	166.1	25.1	42.2	395.0
8	73.84	77.1	84.1	13.1	22.0	213.0
16	37.9	39.4	43.4	7.2	12.1	121.0
32	19.9	20.7	23.2	4.2	7.4	69.0

Arrays may be redistributed across subroutine boundaries. A dummy argument which is distributed differently than its actual argument in the calling routine is automatically redistributed upon entry to the subroutine by the compiler, and is automatically redistributed back to its original distribution at subroutine exit. These operations are performed by the redistribution primitives which transform from *block* to *cyclic* or vice versa.

When a distributed array is passed as an argument to some of the run-time support primitives, it is also necessary to provide information such as its size, distribution among the nodes of the distributed memory machine etc. All this information is stored into a structure which is called *distributed array descriptor* (DAD) [34].

In summary, parallel intrinsic functions, communication routines, dynamic data redistribution primitives and others are part of the run-time support system.

7 Example and performance Results

We use Gaussian elimination with partial pivoting as an example for translating a Fortran90D/HPF program into a Fortran+MP program. The Fortran90D/HPF code is shown in Figure 4. Arrays *a* and *row* are partitioned by compiler directives. The second dimension of *a* is block-partitioned, while the first dimension is not partitioned. Array *row* is block-partitioned. This program illustrates the convenience for the programmer of working in Fortran 90D/HPF. Data parallelism is concisely represented by array operations, while the sequential computation is expressed by *do* loops. More

importantly, explicit communication is not needed since the program is written for a single address space.

Figure 5 shows how the Fortran 90D/HPF compiler translates Gaussian Elimination into Fortran 77+MP form. It is easy to see that the generated code is structured as alternating phases of local computation and global communication. Local computations consist of operations by each processor on the data in its own memory. Global communication includes any transfer of data among processors. The compiler partition the distributed arrays into small sizes and the parallel constructs are sequentialized into a *do* loop.

The compiler generates the appropriate communication primitives depending on the reference pattern of distributed array. For example, the statement

```
temp = ABS(a(:,k))
```

is transformed into a broadcast primitives since the array a is distributed in the second dimension. All runtime routines are classified according to data types. For example, $_R$(Real) specifies the data type of the communication and $_V$ specifies that it is vector communication. The primitive set_DAD is used to fill the Distributed Array Descriptor (DAD) associated with array a so that it can be passed to the *broadcast* communication primitive at run-time. The DAD has sufficient information for the communication primitives to compute all the necessary information including local lower and upper bounds, distributions, local and global shape etc. In this way the communication routines also has an option to combine messages for the same source and destination into a single message to reduce communication overhead. This is the typical characteristic of our compiler since we are only parallelizing array assignments and forall statements in Fortran 90D/HPF, there is no data dependency between different iterations. Thus, all the required communication can be performed before or after the execution of the loop on each of the processors involved.

The intrinsic function *MAXLOC* is translated into the library routine *MaxLoc_R_M*. The suffix $_R$ specifies the data type and $_M$ specifies that *MAXLOC* intrinsic has optional mask array. Once again the array information passed to the run-time system with the associated $_DAD$ data structure.

Several types of *communication* and *computation* optimizations can be performed to generate a more efficient code. In terms of *computation* optimization, it is expected that the scalar node compiler performs a number of classic scalar optimizations within basic blocks. These optimiza-

```
 1.        integer, dimension(N) :: indx
 2.        integer, dimension(1) :: iTmp
 3.        real, dimension(N,NN) :: a
 4.        real, dimension(N) :: fac
 5.        real, dimension(NN) :: row
 6.        real :: maxNum
 7.  C$  PROCESSORS  PROC(P)
 8.  C$  DECOMPOSITION TEMPLATE(NN)
 9.  C$  DISTRIBUTE TEMPLATE(BLOCK)
10.  C$  ALIGN row(J) WITH TEMPLATE(J)
11.  C$  ALIGN a(*,J) WITH TEMPLATE(J)
12.
13.        indx = -1
14.        do k = 0, N-1
15.          iTmp = MAXLOC(ABS(a(:,k)), MASK = indx .EQ. -1)
16.          indxRow = iTmp(1)
17.          maxNum = a(indxRow,k)
18.          indx(indxRow) = k
19.          fac = a(:,k) / maxNum
20.
21.          row = a(indxRow,:)
22.          forall (i = 0:N-1, j = k:NN-1, indx(i) .EQ. -1)
23.     &      a(i,j) = a(i,j) - fac(i) * row(j)
24.        end do
```

Figure 4: Fortran90D/HPF code for Gaussian elimination.

```
B= NN/P
integer indx(N)
real aLoc(N,B)
real fac(N)
real rowLoc(B)
real maxNum
integer source(1)

call grid_1(P)
do i = 1, N
  indx(i) = -1
end do
do k = 1, N
    do i = 1, N
      mask(i) = indx(i) .EQ. -1
    end do
    source(1)=(k-1)/B
    call set_DAD_2(aLoc_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
    call set_DAD_1(temp1_DAD, 1, N, N)
    call broadcast_R_V (temp1, temp1_DAD, aLoc, aLoc_DAD, source)
    call set_DAD_1(temp1_DAD, 1, N, N)
    call set_DAD_1(mask_DAD, 1, N, N)
    indxRow = MaxLoc_1_R_M(temp1, temp1_DAD, N, mask, mask_DAD)
    source(1)=(k-1)/B
    call broadcast_R_S(maxNum, aLoc (indxRow, k-my_id()*B), source)
    indx(indxRow) = k
    call set_DAD_2(a_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
    call set_DAD_1(temp2_DAD, 1, N, N)
    call broadcast_R_V (temp2, 1, temp2_DAD, aLoc, 2, aLoc_DAD, source)
    do i = 1, N
       fac (i) = temp2 (i) / maxNum
    end do
    do i = 1, 10
       rowLoc (i) = aLoc (indxRow, i)
    end do
```

Figure 5: Fortran 90D/HPF compiler generated Fortran77+MP code for Gaussian elimination.

```
        call set_BOUND(k, NN, llb, lub, 1, B)
        do j = llb, lub
           do i = 1, N
              if (indx (i) .EQ. (-1)) THEN
                 aLoc (i, j) = aLoc (i, j) - fac (i) * rowLoc (j)
              end if
           end do
        end do
    end do
```

Figure 5: Fortran 90D/HPF compiler generated Fortran77+MP code for Gaussian elimination (cont.)

tions include common subexpression elimination, copy propagation (of constants, variables, and expressions), constant folding, useless assignment elimination, and a number of algebraic identities and strength reduction transformations. However, Fortran 90D/HPF may perform several optimizations to reduce the total cost of communication. The compiler can generate better code by observing the following:

```
15.          Tmp = ABS(a(:,k))
17.          maxNum = a(indxRow,k)
19.          fac = a(:,k) / maxNum
```

The distributed array section $a(:,k)$ is used at lines 15,17 and 19. The array a is not changed between line 15-19. Because the compiler performs statement level code generation for the above three statements. Each statement causes a broadcast operation. However, the compiler can eliminate two of three communication calls by performing the above dependency analysis. It need only generate one broadcast for line 15 which communicates a column of array a. The Lines 17 and 19 can use that data as well. The optimized code is shown in Figure 6. We generated this programs by hand since the optimizations have not yet been implemented in our compiler. Currently our compiler performs statement level optimizations. It does not perform basic-block level optimizations.

```
 do k = 1, N
     do i = 1, N
       mask(i) = indx(i) .EQ. -1
     end do
     source(1)=(k-1)/B
     call set_DAD_2(aLoc_DAD, 1, N, N, k-my_id()*B, k-my_id()*B, B)
     call set_DAD_1(temp1_DAD, 1, N, N)
     call broadcast_R_V (temp1, temp1_DAD, aLoc, aLoc_DAD, source)
     call set_DAD_1(temp1_DAD, 1, N, N)
     call set_DAD_1(mask_DAD, 1, N, N)
     indxRow = MaxLoc_1_R_M(temp1, temp1_DAD, N, mask, mask_DAD)
     maxNum=temp1(indxRow)
     indx(indxRow) = k
     do i = 1, N
        fac (i) = temp1 (i) / maxNum
     end do
     do i = 1, 10
        rowLoc (i) = aLoc (indxRow, i)
     end do
     call set_BOUND(k, NN, llb, lub, 1, B)
     do j = llb, lub
        do i = 1, N
           if (indx (i) .EQ. (-1)) THEN
              aLoc (i, j) = aLoc (i, j) - fac (i) * rowLoc (j)
           end if
        end do
     end do
end do
```

Figure 6: Gaussian elimination with communication elimination optimization.

To validate the performance of our compiler on Gaussian Elimination which is a part of the FortranD/HPF benchmark test suite [37], we tested three codes on the iPSC/860 and plotted the results. 1-) The code in given 5. This is shown as the dotted line in Figure 7, and represent the compiler generated code. 2-) The code in Figure 6. This appears as the dashed line in Figure 7 and represents the hand optimized code on the compiler generated code as discussed above. 3-) The hand-written Gaussian Elimination with Fortran 77+MP. This appears as the solid line in Figure 7. The code is written outside of the compiler group at NPAC to be unbiased.

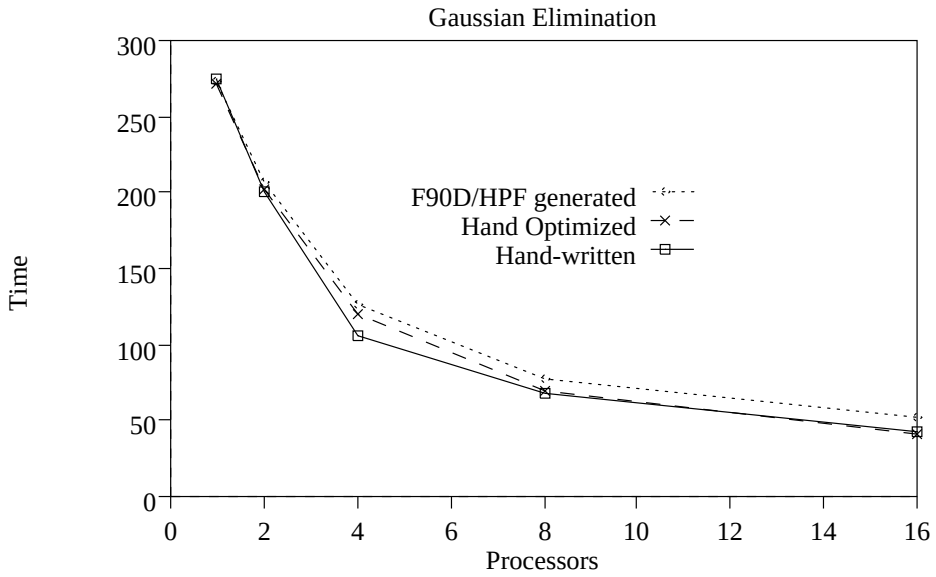

Figure 7: **Performance of three version of Gaussian Elimination. Matrix size is 1023x1024 (time in seconds).**

The programs were compiled by using Parasoft Express Fortran compiler which calls Portland Group if77 release 4.0 compiler with all optimization turned on (-O4). We can observe that the performance of the compiler generated code is within 10% of the hand-written code. This is due to the fact that the compiler generated code produces an extra communication calls that can be eliminated using optimizations. The hand optimized code gives very near performance to hand-written code. From this experiment we conclude that Fortran 90D/HPF compiler which is incorporated with optimizations can compete with hand-crafted code on some significant algorithms, such as Gaussian Elimination.

8 Conclusions

Fortran 90D/HPF are languages that incorporate parallel constructs and allow users to specify data distributions. In this paper, we presented a design for Fortran 90D/HPF compiler for distributed memory machines. Specifically, techniques for processing distribution directives, computation partitioning, communication generation were presented. We believe that the methodology presented in this paper to compile Fortran 90D/HPF can be used by the designers and implementors for HPF language.

Acknowledgments

We are grateful to Parasoft for providing the Fortran 90 parser and Express without which the prototype compiler could have been delayed. We would like to thank the other members of our compiler research group I. Ahmad, R. Bordawekar, R. Ponnusamy, R. Thakur, and J. C. Wang for their contribution in the project including the development of the run-time library functions, testing, and help with programming. We would also like to thank K. Kennedy, C. Koelbel, C. Tseng and S. Hiranandani of Rice University and J. Saltz and his group of Maryland University for many inspiring discussions and inputs that have greatly influenced this work.

References

[1] American National Standards Institue. Fortran 90: X3j3 internal document s8.118. *Summitted as Text for ISO/IEC 1539:1991*, May 1991.

[2] G. C. Fox, S. Hiranadani, K. Kenndy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical report, Rice and Syracuse University, 1992.

[3] High Performance Fortran Forum. High performance fortran language specification version 1.0. *Draft, Also available as technical report CRPC-TR92225 from the Center for Research on Parallel Computation, Rice University.*, Jan. 1993.

[4] The Thinking Machine Corporation. *CM Fortran User's Guide version 0.7-f*, July 1990.

[5] Maspar Computer Corporation. *MasPar Fortran User Guide version 1.1*, Aug. 1991.

[6] G. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-78b, Syracuse University, 1991.

[7] D. Padua B. Leasure D. Kuck, R. Kuhn and M. Wolf. Dependence graph and compiler optimizations. *Proc. of 8th ACM Symp. Principles on Programming Lang.*, September 1981.

[8] D. Kuck D. Padua and D. Lawrie. High speed multiprocessor and compilation techniques. *IEEE Trans. Computers.*, September 1980.

[9] D. Callahan and K. Kennedy. Compiling programs for Distributed Memory Multiprocessors. *The Journal of Supercomputing*, pages 171–207, 1988.

[10] S. Hiranandani, K. Kennedy, and C.W. Tseng. Compiler support for machine-indepentet Parallel Programming in Fortran D. *Compiler and Runtime Software for Scalable Multiprocessors*, 1991.

[11] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimization for Fortran D on MIMD distributed-memory machines. *Proc. Supercomputing'91*, Nov 1991.

[12] Philip Hatcher and Michael Quinn. Compiling Data-Parallel Programs for MIMD Architectures. 1991.

[13] J.H Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. Technical Report SNARC 92-02, Southampton Novel Architecture Research Centre, 1992.

[14] T. Brandes. ADAPTOR Language Reference Manual. Technical Report ADAPTOR-3, German National Research Center for Computer Science, 1992.

[15] J. Li and M. Chen. Compiling Communication -Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 361–376, July 1991.

[16] M. Chen and J.J Wu. Optimizing FORTRAN-90 Programs for Data Motion on Massivelly Parallel Systems. Technical Report YALEU/DCS/TR-882, Yale University, Dep. of Comp. Sci., 1992.

[17] H. Berryman J. Saltz and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, December 1991.

[18] R. Mirchandaney J. Saltz, K. Crowley and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, December 1991.

[19] H. Berryman J. Saltz, J. Wu and S. Hiranandani. Distributed Memory Compiler Design for Sparse Problems. *Interim Report ICASE, NASA Langley Research Center*, 1991.

[20] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE: Transaction on Parallel and Distributed Systems*, pages 179–193, March 1992.

[21] M. Gupta. Automatic Data Partitioning on Distributed Memory Multicomputers. Technical Report PhD thesis, University of illinois at Urbana-Champaign, 1992.

[22] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi Automatic SIMD/MIMD Parallelization. *Parallel Computing*, January 1988.

[23] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. *PPoPP*, March 1990.

[24] C. Koelbel and P. Mehrotra. Supporting Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[25] Z. Bozkus et al. Compiling the FORALL statement on MIMD parallel computers. Technical Report SCCS-389, Northeast Parallel Architectures Center, July 1992.

[26] G. C. Fox, M.A. Johnson, G.A. Lyzenga, S. W. Otto, J.K. Salmon, and D. W. Walker. In *Solving Problems on Concurent Processors*, volume 1-2. Prentice Hall, May 1988.

[27] M. Wu and G. Fox et al. Compiling Fortran 90 programs for distributed memory MIMD paralelel computers. Technical Report SCCS-88, Northeast Parallel Architectures Center, May 1991.

[28] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, pages 102–118, Feb 1990.

[29] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, pages 213–221, Oct 1991.

[30] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S.H Tseng. Automatic Array Alignment in Data-Parallel Programs. *Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1993.

[31] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, September 1990.

[32] R. Das, J. Saltz, and H. Berryman. A Manual For PARTI Runtime Primitives. *NASA,ICASE Interim Report 17*, May 1991.

[33] A.V. Aho, R. Sethi, and J.D Ullman. *Compilers Principles, Techniques and Tools.* March 1988.

[34] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Fortran 90D Intrinsic Functions on Distributed Memory Machines: Implementation and Scalability. Technical Report SCCS-256, Northeast Parallel Architectures Center, March 1992.

[35] ParaSoft Corp. *Express Fortran reference guide Version 3.0*, 1990.

[36] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam. A Users Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[37] A. G. Mohamed, G. Fox, G. V. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Application Benchmark Set for Fortran-D and High Performance Fortran. Technical Report SCCS-327, Northeast Parallel Architectures Center, May 1992.

[38] Z. Bozkus et al. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. Technical Report SCCS-444, Northeast Parallel Architectures Center, 1993.